

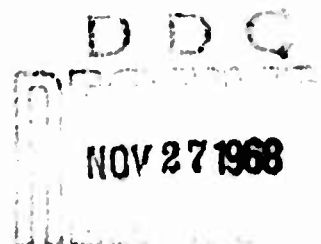
Reproduced by the
CLEARINGHOUSE
for Federal Scientific & Technical
Information Springfield Va. 22151

**SOME TECHNIQUES FOR ALGORITHM OPTIMIZATION
WITH
APPLICATION TO MATRIX ARITHMETIC EXPRESSIONS**

by

Robert Alan Wagner

**Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
June 27, 1968**



**Submitted to the Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy**

**This work was supported by the Advanced Research Projects
Agency of the Office of the Secretary of Defense (SD-146) F-44620-67-C-0058
and is monitored by the Air Force Office of Scientific
Research. Distribution of this document is unlimited.**

ABSTRACT

Algorithm optimization can be accomplished by an exhaustive search over alternative algorithms for performing some programming task. The resulting algorithms are optimum only with respect to a program technology--the particular set of alternatives investigated. Thus, larger program technologies can be expected to yield better algorithms. This thesis contributes to the production of optimum algorithms in two ways. First, a technique ("loop-fusion") was developed for producing new algorithms equivalent to old algorithms, and thus expanding program technologies. Second, a technique ("comparison") is described which reduces the effort required by certain exhaustive searches over "well-structured" search spaces. These techniques are applied to the production of algorithms for evaluating matrix arithmetic expressions (MAE). (The operators, + and *, in such arithmetic expressions are interpreted as matrix addition and multiplication, respectively.) A method is described for producing, for any MAE, an algorithm for its evaluation which requires fewest arrays for holding N by N matrices, while not requiring more execution time than the "standard" MAE evaluation algorithm. Although the algorithm-production method used is basically an exhaustive-search over a large space of program alternatives for each subexpression of the given MAE, the effort this method requires grows only linearly with the number of operators in the given expression.

ACKNOWLEDGMENTS

I wish to thank my adviser, Professor Alan J. Perlis, for his efforts to aid me in this work. He suggested the specific application of this thesis. He encouraged me to continue my effort when all seemed hopeless. Furthermore, without his criticism of the presentation, this thesis would probably be totally unreadable. I also wish to thank Miss Sally Dewald, who suffered through many pages of nearly indecipherable handwriting to produce these typed pages.

TABLE OF CONTENTS

TITLE PAGE.	1
ABSTRACT.11
ACKNOWLEDGMENTS.111
TABLE OF CONTENTS.	iv
CHAPTER I.	1
I.1 General Problem of Program Optimization.	1
I.2 A Specific Problem.3
I.3 Prior Work.	12
I.4 Statement of the Problem.	16
I.5 Overview of Our Approach.	17
CHAPTER II.19
II.1 Basic Definitions.	19
II.2 Parse-Trees and Expressions.	20
II.3 Parse-Tree Examples.	21
II.4 The " n^3 " Elementary Algorithms.22
II.5 Loop Fusing.	25
II.6 Fusion in Flow Charts: Graphic Description.29
II.7 Loop Fusion Conditions.30
II.8 Storage Savings in Loop Fusion.45
II.9 Summary of Loop Fusion Conditions.	48
II.10 Algol and Flow-Chart Language.50
II.11 Algorithm Fusion (parallel connection).	50
II.12 Matrix Operation Algorithms (MOA's).52
II.13 Shapes, the "Valences" of an MOA.	56
II.14 Explicit Rule for Developing Shapes for Arrays used in Algol Programs.	59
II.15 Parallel Connection Algorithm.60
II.16 Matrix Elementary Algorithms.	64
II.17 Canonical k-MEA's.67

CHAPTER III.	69
III.1 Elementary Expression Parse-Trees (EEPT's).	69
III.2 Alg-Tree Definitions.	70
III.3 Major Properties of Alg-Trees.	74
III.4 Result-Array and Fringe-Set Array Storage Overlap.	76
III.5 Calculating Intermediate 2-Array Requirements of an Alg-Tree	79
III.6 The Leaves-In Algorithm.	84
III.7 Effort Estimates Motivating Search Reduction.	91
CHAPTER IV.	97
Introduction.	97
IV.1 The Fv-Set Comparison Theorem	102
IV.2 Applications of the Comparison Theorem to the Leaves-In Algorithm.	109
IV.3 The Leaves-In Algorithm, with Comparison Theorem.	111
IV.4 Leaves-In Algorithm Effort Requirement.	112
CHAPTER V.	120
V.1 Summary of Results...	120
APPENDIX I.	126
1. Winograd's Matrix Multiply Algorithm	126
2. Additional Shapes Defined for Winograd's Algorithm.	127
3. Variations on Winograd's Algorithm (N Even).	127
APPENDIX II.	130
Leaves-In Algorithm in APL.	130
External Representations.	131
Detailed Examples.	133
BIBLIOGRAPHY	154

CHAPTER I

I.1 General Problem of Program Optimization

Problems are often presented to a human programmer in ways which allow a multitude of possible approaches to their solution. The programmer must then decide on some basis, which possible approach should be used. Furthermore, a given "program" is, if it is at all useful, normally destined to be a sub-program of various larger programs. One has no reason to hope that the implementation of the sub-program is the same in the optimum implementation of each program in which it is embedded. Thus, a program cannot actually be optimized permanently in isolation. The programmer must optimize it for the particular context in which it is to be used.

One form of the program-optimization problem can be stated as follows:

Given a programming task formulated as a desired transformation or mapping of input data to output data, find that program which "best" implements the task.

A program specifies the sequence of operations some processor is to perform in order to accomplish the desired transformation. Each operation is itself a transformation, which must be drawn from a fixed set of possible operations, the repertoire of instructions of the given processor. When more than one sequence of operations can be used to accomplish a given programming task, such sequences are termed equivalent, as are the programs which specify them. That program is "best" which, of all equivalent programs, minimizes some program cost function.

Programs, in their specification of operation sequences, can be associated with "costs". These costs need not be monetary. In general, they represent the amount of some scarce resource used in creating, or executing a given program. Examples of costs include:

- (1) Programmer's time required in creating a program
- (2) Processor time used in executing some program

- (3) Processor memory space required before the program can be executed.

Costs (2) and (3) are of particular interest. These costs describe the performance of the final product--the program. We therefore define a program cost function to be some non-decreasing function of the processor time expended, and memory space required, during the program's execution.

In fact, a class of optimization problems can be devised, depending on the precise description of the program cost function. For example, one can conceive of an environment in which total main memory space is limited. Then no program is acceptable which requires more main memory than this amount. Among the equivalent programs which are acceptable, some require least processor execution time. These would then be preferred. In place of a single function of memory space and execution time, one of the variables enters the optimization problem in a constraint, while the other makes up the function to be optimized.

An equally valid description of the program cost function can reverse the roles "space" and "time" played in the previous example. That is, we conceive of a situation wherein constraints are placed on the execution time of some programming task, leaving us to choose among the equivalent programs for this task one which uses least memory space. Such a situation arises in certain "multi-programmed" computer systems, which employ the physical memory allocation technique called "paging". In these environments, potentially, large amounts of space are available, at increasing cost in "response time". It would seem desirable in such an environment to choose a program which uses least space, while requiring the processor time required for execution to lie below some upper limit.

The cost functions described here all depend on the measurement of a program's execution time and space requirements. These requirements depend on both the program and the particular input data with which that program is supplied in a given execution. In general, a program is written to apply to many different selections of input data. It seems

desirable to discuss its requirements for several such input data sets at once. One convenient way to describe program behavior for large classes of possible inputs is to "parametrize" that program's requirements--to express the space and time requirements in terms of certain "characteristic numbers" derivable from the data.

Thus far, the discussion of optimization of programs has implied a search over all possible programs which specify a given programming task. Unfortunately, for many programming tasks we know of no way to characterize all programs which specify that task. Nevertheless, methods of improving programs are still desirable.

For certain programming tasks, a number of alternative programs are known. A search can be performed which is limited to a set of programs for a programming task which are derivable in some specific ways. The best program among those considered will be termed "optimum with respect to some (specified) technology", or technologically optimum, for short. Technologically optimum programs may well yield near-optimum values for the program criterion function. Additional improvements can be realized by increasing the number of programs derivable, that is, by expanding the technology.

I.2 A Specific Problem

The general problem of program optimization tends to founder on the problem of programming task representation. As presented to a human programmer, many if not most programming tasks are not well-defined. Not only does the programmer often have great scope in choosing solution techniques; often he may choose the characteristics of the solution as well. Such freedom limits the ability of computer optimization techniques to derive equally satisfying results. Because the limits of acceptability of programs are vague, and indeed only informally stated, solutions proposed by algorithms cannot be tested for acceptability.

Several classes of well-specified programming tasks do exist, however. Each "higher-level" language construct, such as the expression in Algol, or the DO-loop of FORTRAN, specifies a programming task some-

what independently of specific sequences of instruction on a specific machine. The semantics, or meaning, of instances of such constructs thus allow more than one program to correctly specify that meaning. Furthermore, because many instances of each construct may be presented to an optimization (program-choice) algorithm, it is profitable to derive such algorithms. The present work describes a method, based ultimately on an exhaustive search over programming alternatives, for "compiling" or "translating" one high-level language construct: the "matrix arithmetic expression". Several authors have advocated the addition of matrix arithmetic capabilities to various programming languages. The matrix arithmetic expression provides a basic construction for specifying such arithmetic.

The syntax of a matrix arithmetic expression can be taken to be that of an Algol expression whose \langle variables \rangle are all \langle array identifiers \rangle , and whose operators are restricted to '+' and '*'. In these expressions, + and * designate matrix addition and multiplication.

We can successfully compile technologically-optimal programs for instances of a sub-class of all expressions. Our optimization algorithm requires that:

- (1) All variables of the expression must be N-by-N (square) arrays;
- (2) The expression must be "fully parenthesized"; and
- (3) The expression may not contain common subexpressions.

A fully parenthesized expression syntactically describes exactly one decomposition of the expression into one-operator subexpressions. Thus, we will make no attempt to employ the associative and distributive laws of matrix algebra to derive equivalent expressions. A common subexpression is a subexpression, more than one instance of which occurs in the expression. Thus 'A+B' is a common subexpression of the expression

$$(A+B) * (A+B).$$

We will also assume that the matrices we deal with are "general", so that no special space-saving storage techniques are possible.

We will describe a particular "technology" for programs which eval-

uate matrix arithmetic expressions. From this technology, an expanded technology can be developed, using a technique which may well be useful for creating new equivalents to programs for other programming tasks. A search-procedure is developed over the programs in the expanded matrix arithmetic expression technology. This search-procedure accepts only those programs whose time-requirement is no greater than that required by the "standard" technology. It searches for a program whose memory-space requirement is least. The search-procedure is shown to require compile-time computation resources which increase exponentially with the number of operators in the given expression. Finally, a general technique for reducing the effort of any "structured" exhaustive search is developed, and applied to the present search, reducing compile-time effort to linear dependence on the number of operators in the given expression.

From the (informal) semantics of matrix arithmetic expressions, it should be clear that techniques known for "compiling" scalar expressions are applicable to matrix arithmetic expressions. Techniques are available which "compile" arbitrarily complex scalar expressions into instances of a small number of basic assignment statements. For example, basic assignment statements for scalar arithmetic expressions with the syntax of matrix arithmetic expressions are:

$$A \leftarrow B + C \quad \text{and} \quad A \leftarrow B * C.$$

A compilation technique based only on the syntax of the given expression can resolve any expression into a sequence of systematic substitution instances of the above assignments. (A systematic substitution replaces each variable name in an assignment with new names, chosen so that the new name for the left-side variable does not agree with the new name of any other right-side variable. Thus,

$$A \leftarrow X * Y \quad \text{and} \quad Z \leftarrow Q * W$$

are systematic substitution instances of $A \leftarrow B * C$, but

$$A \leftarrow A * X \quad \text{and} \quad Z \leftarrow Q * Z$$

are not.)

New variables can be chosen to hold the values of subexpressions

until these values are input to a later assignment. Thus, the functional composition of the binary addition and multiplication functions making up the given expression can be achieved. Matrix arithmetic expressions can be compiled in exactly the way scalar expressions are compiled, yielding sequences of systematic substitution instances of the (syntactically) same basic assignment statements. For each of the basic matrix assignment statements, an algorithm can be devised. Thus, a compilation into basic assignments serves to produce an algorithm for computing the expression.

In the case of scalar expressions, relatively little memory space is required to hold each intermediate result. Accordingly, many compilers make no attempt even to re-use variables used for intermediate results.

Matrix arithmetic presents some motivation for space-optimal compilation. In compiling matrix arithmetic expressions using this technique, a set of N^2 variables must be allocated to hold each intermediate result. Because N may well be large, a significant amount of memory could be demanded by the compiler for intermediate matrices. Accordingly, some effort by the compiler to reduce the storage space it allocates for the compiled program is desirable.

In the discussion of the general problem of program optimization, it was noted that, realistically, program time and space requirements should be parametrized. Both the time and space required for computing matrix arithmetic expressions can be regarded as functions of N , where each matrix entering the expression is N -by- N . Thus, each set of variables capable of holding a matrix (called a 2-array) must include N^2 variables. The time requirement for computing

$$A \leftarrow B * C$$

can be stated as: N^3 additions, and N^3 multiplications, when one algorithm is used, or

$$3N^3/2 \text{ additions, and } N^3/2 \text{ multiplications}$$

when an algorithm recently discovered by S. Winograd [14] is employed.

In general, time and space requirements for computing a matrix arithmetic expression are polynomials in N . For large enough N , the leading term of a polynomial in N dominates the polynomial, in the sense that the contribution of all other terms are negligible with respect to it. (The leading term of a polynomial in N is that term in which N has the largest exponent.) Accordingly, we will approximate time and space requirements by the leading term of their representation as polynomials in N .

The space-requirement for a program to calculate a given matrix arithmetic expression has several components, each corresponding to a term in the polynomial in N . Since we have stated that certain terms of the polynomial will be ignored in our optimization, it seems worthwhile mentioning the program entities to which they correspond.

The leading term of the space-polynomial clearly counts the number of 2-arrays required. The linear term in N measures the number of 1-arrays, each capable of holding a 1-by- N or N -by-1 matrix, or vector, of values. The program itself is represented in memory at execution time. However, its size is independent of N , since we will implement it by means of "loops". The program-size thus enters the constant term of the space polynomial. Thus, the space requirement for a matrix arithmetic expression's evaluation is dominated by the number of 2-arrays needed for that evaluation.

We will seek a technologically-minimum-space program. The program-class we search will include only programs whose time-requirement is smaller than a time-standard, derivable from the given expression.

Two methods for computing matrix multiplication have been alluded to. One the set of N^3 algorithms, requires N^3 scalar additions and multiplications to perform a matrix multiplication; the other, the set of $N^3/2$ algorithms, requires $3N^3/2$ additions and $N^3/2$ multiplications. Suppose all basic matrix multiplication statements in the "standard" compilation of some expression is implemented by the same algorithm, and that no element of any subexpression is recomputed during the expression's evaluation. Then the number of scalar additions and mul-

tifications needed for this no-unnecessary-computation implementation forms a reasonable upper bound on the time requirement of an "acceptable" program.

We will seek a program for evaluating a given matrix expression whose space-requirement is least, subject to the requirements that the program

- (1) be generatable by techniques to be presented, and
- (2) requires no more time than a no-unnecessary-computation sequence of one-operator basic assignments.

Programs which satisfy (2) are termed minimum-connection-time programs. We are seeking a program whose space requirement is least, where we approximate a program's space requirement by the number of 2-arrays it uses. The 2-array requirements of two programs will thus be compared in the course of the search outlined. However, we can show that certain 2-arrays are needed by any program to evaluate a given matrix expression. These 2-arrays hold the matrices which are input to the expression. These variables must remain present and undisturbed throughout the expression's evaluation. Thus, input 2-arrays do not affect the comparison of two programs. In effect, only non-input 2-arrays need be counted in the program criterion function. These non-input 2-arrays will be termed "intermediate 2-arrays".

We will demonstrate an expansion of the basic compilation technology which will introduce more "basic" matrix assignment statements. These statements will contain more than one operator. In fact, they are derived by substituting the expressions of basic assignment statements for the variables in other assignments. Corresponding to each new assignment, we will show how an algorithm, called a matrix elementary algorithm, (MEA), can be constructed, having the following properties:

- (1) Its time requirement is the same as that of the sequence of basic one-operator assignments from which its expression was derived;
- (2) It requires only $k \cdot N$ intermediate variables for its evaluation. Here, k does not depend on N .

These algorithms are created from sequences of basic (1-operator) assignment algorithms by a process called "loop-fusion". Basically, this process allows small portions of the matrix which represents the value of a subexpression to be computed, and then used at once in computing a portion of the expression enclosing that subexpression. This portion of the intermediate result need not be retained longer. The variables used to hold this part of the intermediate result can then be used to hold another portion of the intermediate result. The fusion may thus require as few as $k \cdot N$ intermediate variables.

The technique of loop fusion may permit combining two loops which are not part of matrix arithmetic algorithms. One of the principal results of this thesis is a set of sufficient conditions under which two loops may fuse into one computationally equivalent loop. Any technique for generating equivalent programs increases the set of programs over which a technologically optimizing algorithm may search. Thus, loop fusion holds potential for improving programs for tasks other than evaluation of matrix arithmetic expressions.

Loop fusion permits generation of a potentially infinite number of matrix elementary algorithms (MEA's). However, the syntax of their associated expressions is more restrictive than the syntax of matrix arithmetic expressions. Not every matrix arithmetic expression can be evaluated using a single matrix elementary algorithm. As a result, techniques are needed for deciding just which MEA's should be used to evaluate each subexpression of a given matrix arithmetic expression.

The optimum decomposition of matrix arithmetic expressions into expressions which MEA's are capable of evaluating can not be decided apart from the given expression. In other words, no one MEA is obviously better than another, for all expressions. For example, it might be supposed that the larger the expression evaluable by an MEA, the better that MEA is. The intermediate variable requirements of a "large" MEA are no worse than that of a smaller MEA. In some sense, the overhead of computing the large expression can be apparently allocated over more operators, reducing the per-operator storage costs. Unfortunately,

large-expression MEA's have another property which limits their usefulness: every input of an MEA must be present simultaneously. Thus, a large-expression MEA whose inputs are all intermediate results requires more intermediate 2-arrays to be present than a similar small-expression MEA. Nonetheless, when a large-expression MEA's inputs correspond primarily to inputs to the given expression, its use is desirable.

An algorithm, called the "leaves-in algorithm", was devised for generating all the possible decompositions of a given matrix arithmetic expression into MEA's. This algorithm is "efficient" in the sense that it never re-generates an MEA used to evaluate a particular subexpression, as the evaluation rules for other subexpressions are varied. Instead, all possible MEA-decompositions which can be used to evaluate a sub-expression are retained in memory, and combined with the MEA-decompositions for evaluating other subexpressions to generate new MEA-decompositions. Unfortunately, the "effort" (computation time) required by this algorithm was found to grow exponentially with the number of operators in the expression, for certain expressions. This potentially large effort is undesirable, since it makes the cost of obtaining a technologically optimum program unreasonably large.

A general technique for reducing the effort required by certain exhaustive researches was devised, and applied to the leaves-in algorithm. The technique is not "heuristic", in that no chance of missing an optimum solution is introduced by its use. Furthermore, the technique may well be useful in reducing the effort required by other exhaustive search procedures.

Roughly, in any exhaustive search, "states" of the search are produced. Often, not all variables in the state-vector are computed simultaneously. We can speak of a "partial state", which represents the situation obtained when not all state variables are given values. A partial state may lead to any of a large number of complete states, depending on the assignments made to the variables not assigned values in the partial state. A particular set of values assigned to the non-partial-state variables will be termed a "completion".

Most of the effort in an exhaustive search involves generating all the possible completions of each partial state. Now suppose two partial states are known such that the same variables are fixed (to different values) in each, and such that any completion of one is a possible completion of the other. Thus, if A and B are partial states, and C is a given "completion", if $A \cup C$ (read "A completed by C") is valid, so is $B \cup C$. Also, suppose $N(S)$ is the value of a state, and we seek a state of minimum value. If for all completions C, $N(A \cup C) \geq N(B \cup C)$, then completions of partial-state A need not all be examined. For every complete state $A \cup C$ generated by any completion C, there is a complete state $B \cup C$ generated by that same C which is better. Now, notice that the statement

$$(1) \quad \forall C [N(A \cup C) \geq N(B \cup C)]$$

is a predicate on A and B which is independent of C. If we can discover a predicate equivalent to (1) whose evaluation does not require the generation of all possible completions C, we can compare partial states using it. The resulting algorithm may reduce the number of states generated tremendously.

The power of the technique described depends on several properties of the space searched, and the variables chosen to describe states in that space. In some searches, the comparison may lack "power"--for example, it may only hold between identical partial states. In other searches, few pairs of partial states which yield true for the value of the comparison may ever be generated. Nonetheless, in some searches over "well-structured" spaces, such comparisons may drastically reduce the search effort.

In the search for the best MEA-decomposition of a given matrix arithmetic expression, the comparison theorem proved quite useful. Here, a "partial state" corresponds to a particular decomposition of one subexpression. A "state" corresponds to a particular decomposition of the entire given expression. A partial state may be completed by any decomposition of some subexpression not part of the partial state's

subexpression. States arise from fusions of the MEA's used in computing subexpressions. The evaluation rule for states depends on components in the partial state and in the completion.

The evaluation function of an MEA depends on the set of subexpressions whose values are inputs to this MEA. The number of intermediate 2-arrays needed to compute each input is used to determine the number needed to compute the MEA's result. An MEA A which is extended by loop fusion becomes an MEA, $A \cup C$, whose inputs include all inputs of A , as well as additional inputs, C . These new inputs, adjoined to the input sets of two different algorithms, may completely change the relative space-efficiency of the algorithms.

We were able to discover just when two MEA-decompositions for evaluating some subexpression were interchangeable. By interchangeable, we mean that any valid completion of one is a valid completion for the other. Furthermore, we were able to discover the evaluation-rule, $N(S)$, for MEA-decompositions S . Also a predicate equivalent to (1) for this evaluation rule was discovered. The application of this comparison predicate to the leaves-in algorithm reduces the effort required to a value proportional to, rather than exponential with, the number of operators in the given expression.

I.3 Prior Work

Several aspects of the prior art should be discussed. Some results have been published relating to the general problem of program optimization. Also, various authors have attacked specific problems in this area. We freely admit to being influenced by their approaches. Some previous work has been directed at the production of optimum compilations for scalar-expressions, work whose basic techniques we build on. Finally, some work on optimum compilations of matrix arithmetic expressions has been published, and should be mentioned here.

One of the major forerunners of the approach we employ here seems to have been Simon's "Heuristic Compiler" [9]. Simon's work appears to

have as its goal the production of some program to accomplish a given programming task; however, he appears to have been one of the first to describe a wide variety of programming tasks in such a way that a space of program alternatives for their accomplishment could be visualized. His "before-and-after" description of procedure operations forms such a "state description" of programs. Indeed, he explicitly mentions the possibility that "there will generally be many programs (not all equally efficient or elegant) that will do the same work" [9, pg. 6]. This very naturally suggests a search for the most elegant, or efficient.

Several authors have attacked problems in the general area of optimizing the compilation of specific language-constructs. Notably, Reinwald and Soland [7,8] have discussed at length the problem of converting "Decision tables" into optimal computer programs. Interestingly, they adopt an approach based on an exhaustive search over certain program variations. They advocate use of "branch and bound" techniques for reducing the space searched. Furthermore, they suggest that the space of programs they search exhausts the space of all programs which can be said to be "translations" of a given decision table. Thus, the programs they produce are claimed to be time-optimal, or space-optimal, and they even propose means for locating optimal programs whose criterion function is a linear combination of memory space and execution time.

Another group of problems has been attacked by Winograd [11,12,13]. Winograd treats both problems of designing minimal-time hardware for performing certain computer instructions, and that of designing minimal-operation-cost algorithms for performing certain operations. For the most part, Winograd concerns himself with deriving theoretical lower bounds on the "time" required for certain computer operations. In fact, he usually also demonstrates procedures which yield near-minimal-time operations. Although his approach is not constructive, nevertheless his search for theoretical lower-bounds on quantities we attempt to minimize is certainly relevant. Indeed one of his results (in [14]) directly concerns matrix multiplication. Here, he presents an algorithm for calculating the dot-product of two vectors which, when applied to matrix

multiplication, reduces the number of multiplications required from n^3 to approximately $n^3/2$. We present a derivation of this result in Appendix I, together with algorithms which implement it, and which can be used as "basic algorithms" for the matrix assignment $A \leftarrow B * C$.

Of more direct relevance to the compilation of matrix expressions is an algorithm developed for space-optimal compilation of scalar arithmetic expressions, and described by I. Nakata [5]. This algorithm is based on an analysis of an expression into a data-flow diagram, a precedence-graph showing the necessary time-sequence of subexpression computation. This structure inspired the analysis of the leaves-in algorithm. Nakata's algorithm which we describe briefly here, produces a linear order for the evaluation of subexpressions. This order, of all possible evaluation orders, uses fewest intermediate variables.

Let x be a node in the parse-tree T of an expression E . Suppose $n(x)$ represents the minimum number of intermediate variables needed in computing the subexpression whose sub-tree is rooted at x . Apply the following algorithm to each node of T , applying it to every descendant of a node y before y .

1. If x is a leaf of T , x 's subexpression is a variable input to E , and needs no algorithm for its computation. Set $n(x) = 0$.
2. If x is not a leaf of T , let its immediate descendants be x_1 and x_2 . Then $n(x_1)$ and $n(x_2)$ have already been computed.
 - a. If $n(x_1) > n(x_2)$, ($i, j = 1, 2$), compute x_1 's subexpression first. One cell retains the result of this computation during the following calculation of x_2 's subexpression. Set $n(x) = n(x_1)$, since enough cells remain of the $n(x_1) - 1$ to compute x_2 , which needs only $n(x_2) \leq n(x_1) - 1$.
 - b. If $n(x_1) = n(x_2)$, then regardless of which subexpression is computed first, one cell is required to hold its result. Then an additional $n(x_1)$ cells is needed in computing the other result. x 's subexpression can be

computed, using $A \leftarrow A \text{ op } B$, into one of the cells now holding a subexpression's result. Hence, $n(x) = n(x_1) + 1$.

The operation of this algorithm depends on the presence of elementary operations for performing assignment statements like $A \leftarrow A \text{ op } B$, where one input variable is replaced by the assignment's result. Otherwise, this same algorithm can be extended directly to matrix arithmetic expressions. We show later that operations very similar to

$A \leftarrow A * B$, and of course $A \leftarrow A + B$

are available for matrices A and B .

Galler and Perlis [4] were early advocates of the addition of matrix arithmetic capability to compiler languages. They comment on the potential danger of allowing a compiler to allocate large amounts of storage for the evaluation of matrix expressions. They suggest a number of elegant devices for performing various matrix manipulations. One of particular elegance seems to be their proposal for implementation of the matrix transpose operation. No instructions are needed at execution time. Instead, in each algorithm compiled, the compiler interchanges the indices in the subscript positions of each subscripted variable which is a transposed matrix.

Galler and Perlis also present an interesting technique for computing a succession of matrix products. Indeed, this technique demonstrated the computation of subsets of the elements of a matrix result, followed by immediate use of those elements. We generalize this notion to that of "loop fusion", applicable to algorithms other than matrix arithmetic algorithms, in the sequel.

Galler and Perlis' algorithm for matrix multiplication:

Suppose we wish to compute $A * B * \dots * K$, a product of matrices.

Let x^i represent the i th row of matrix X .

Then we can compute

$$A^i * B = (A * B)^i$$

without using more than one vector of storage. By extension, only vectors of storage are needed in computing

$$(A * B * \dots * K)^1 \\ \text{by } (A * B * \dots * J)^1 * K$$

By repeating this computation of one row-vector of the product for different rows A^1 , the entire product can be produced, using only one matrix to hold the result.

[Galler and Perlis also show that, in the repeated products of row-vectors with matrices needed to produce a result row, only two vectors of storage are needed, at most.]

While the Galler-Perlis algorithm produces highly acceptable programs for computing certain expressions, it is inapplicable to others. For example, it does not apply to: $A * (B + C) * D$. Were we to compute $A^1 * (B + C)$, we could use an entire matrix to hold the value of $(B + C)$. Otherwise, the value of $(B + C)$ would necessarily have to be re-computed as each of the N rows of A were multiplied by $(B + C)$. The re-computation produces a program which is not minimum-connection-time, and is hence unacceptable.

I.4 Statement of the Problem

We consider matrix arithmetic expressions (MAE's) (as distinct from MEA's) whose syntax is:

$$\langle \text{MAE} \rangle :: = \langle \text{MAT} \rangle \mid \langle \text{MAT} \rangle + \langle \text{MAE} \rangle$$

$$\langle \text{MAT} \rangle :: = \langle \text{MAP} \rangle \mid \langle \text{MAP} \rangle * \langle \text{MAT} \rangle$$

$$\langle \text{MAP} \rangle :: = (\langle \text{MAE} \rangle) \mid \langle \text{matrix identifier} \rangle$$

The operators $+$ and $*$ signify matrix addition and multiplication respectively. A $\langle \text{matrix identifier} \rangle$ is declared as an $\langle \text{array} \rangle$ [6].

We restrict the MAE's we investigate as follows:

- (1) Each $\langle \text{matrix identifier} \rangle$ is declared to have subscript bounds of $[1:N, 1:N]$. Thus, each matrix must be square;

- (2) The semantics of an $\langle \text{MAE} \rangle$ or $\langle \text{MAT} \rangle$ which includes more than one operator is interpreted to be right-associative. Thus, we assume that the MAE $A * B * C$ must be computed as: $(A * (B * C))$.
- (3) No $\langle \text{MAE} \rangle$ may contain more than one instance of the same (sub) $\langle \text{MAE} \rangle$.

We propose to find a program which, among a certain set of programs, computes any given MAE using the smallest number of intermediate 2-arrays, subject to a restriction on the acceptability of programs:

Each acceptable program must be a minimum-connection-time program.

The set of programs we study consists of sequences of instances of basic matrix assignment algorithms. The basic matrix assignment algorithms are a potentially infinite collection of algorithms derived from algorithms for the matrix assignments $A \leftarrow B * C$ and $A \leftarrow B + C$. Because the set of basic matrix assignment algorithms is far larger than the collection of algorithms usually used for compilation, the technological space minimum we obtain using them is smaller than that obtainable using only algorithms for $A \leftarrow B * C$ and $A \leftarrow B + C$. However, we can make no claim to have discovered either time or space optimal algorithms, for we have no proof that we have exhausted all programming possibilities in constructing the particular set of programs studied.

1.5 Overview of Our Approach

We first describe a technique, called "loop fusion", for creating new programs equivalent to certain given programs. This technique produces programs which are computationally equivalent to the given programs, and which require the same (or slightly less) execution time. Their patterns of accessing and computing data are different, however.

Using loop-fusion, we find we can grow a potentially infinite collection of algorithms for evaluating matrix expressions. These algorithms, called MEA's, are grown from only five basic algorithms. They each require internal intermediate variables proportional in number

to N.

A compilation algorithm, called the "leaves-in" algorithm is presented next. This algorithm discovers the space-minimal decomposition of a given expression into MEA's. It does so by "tailoring" MEA's to fit each subexpression of the given expression, in all possible ways. While it succeeds in avoiding redundant re-decomposition of sub-branches, it requires computational effort (time) which grows exponentially with the number of operators in the given expression.

A general technique, called "comparison", is proposed to reduce the computational effort of exhaustive search optimization. This technique attempts to avoid generating all possible "completions" of a partially-specified search state. It does so by comparing two interchangeable partial states in all possible completions, without actually generating these completions. By generalizing over all completions, a predicate independent of any completion is produced, which compares two partial states. Certain evaluation rules for states permit derivation of an equivalent predicate which does not mention completions, and may hence be evaluated by examination of only the partial states it compares.

Comparison is applied to partial states in the leaves-in algorithm. Here, a partial state corresponds to a possible algorithm for use in computing a single subexpression. A complete state is an algorithm for computing the entire given expression. By eliminating many partial states as soon as they are generated, the effort-requirement of the leaves-in algorithm is reduced to a linear function of the number of operators in the given expression.

CHAPTER II

II.1 Basic Definitions

The input set of an algorithm is the set of variables whose contents just before the algorithm is executed are accessed during the algorithm's execution.

The change-set of an algorithm is the set of variables stored into during the algorithm's execution.

The result-set of an algorithm is the set of variables which are (1) in the change-set of the algorithm, and (2), whose contents immediately after the algorithm's execution is input to some other algorithm.

The intermediate set of an algorithm is the set of variables in the algorithm's change-set, and not in its result-set.

The inputs to an algorithm are the values the variables in the algorithm's input-set hold just before the algorithm's execution. Similarly, the result of an algorithm is the set of values its result-set holds just after the algorithm's execution.

The word "algorithm" here may be taken to mean "statement-sequence", including the sequence consisting of exactly one statement. We will therefore use, for example, "result-set of a statement" in its sense as defined here.

The word array is our name for the Algol [6] <subscripted variable>. A k-subscript array corresponds to an Algol subscripted variable having k subscript positions.

For our purposes, a k-subscript array is a named set of variables. If two arrays A and B have different names, so

$$\text{name}(A) \neq \text{name}(B)$$

then $A \cap B = \emptyset$, i.e., A and B have no variable in common. Furthermore, if [i] and [j] are two k-tuples such that $[i] \neq [j]$, then $A[i]$ is some particular variable, a member of A, and $A[i] \neq A[j]$. Thus, different

combinations of the k subscripts select different variables, all members of the array.

A simple algorithm is a sequence of loops and assignment statements containing no branches outside loops.

Two simple algorithms are adjacent just when every statement of one precedes every statement of the other, and when no statements intervene between the last statement of the first algorithm, and the first statement of the second algorithm.

If A is a set, the number of elements of A will be denoted size(A).

II.2 Parse-Trees and Expressions

We often find it convenient to refer to expressions by their parse-trees. The parse-tree of an expression is a directed graph with labeled edges. The nodes of this graph correspond to the operators and variables in the expression in such a way that, if the expression is $\langle E1 \rangle \langle OP \rangle \langle E2 \rangle$, the parse-tree contains a node i whose name is the same as $\langle OP \rangle$, and whose left son is a subtree corresponding to $\langle E1 \rangle$, and whose right son is a subtree corresponding to $\langle E2 \rangle$.

Left (right) sons are located by following the branch labeled left (right) to the node it is incident on. We often use "family tree" terminology when dealing with trees. Other terminology used is:

"leaf" - A leaf of a tree has no descendants.

"root" - The root of a tree is a unique node having no ancestor.

"result" - The result of a node X of a parse-tree is the value of the subexpression represented by the subtree rooted at X .

Each of our parse-trees has a root, the node corresponding to its expression's main connective. The expression this operator is a part of is a subexpression of no larger expression. Hence, the node which corresponds to it has no ancestor. Any given algorithm which computes

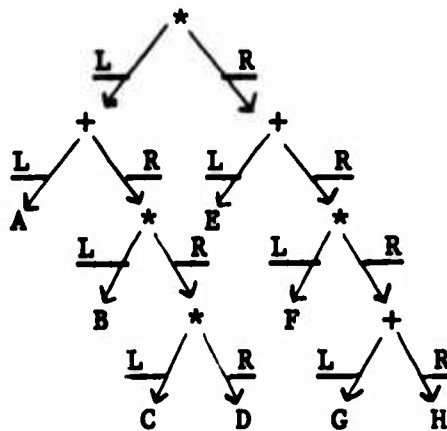
an expression has an associated parse-tree, that of the expression computed by the algorithm. In addition, we attach to its parse-tree characteristics of the algorithm which will enable us to tell when and how algorithms can combine. In particular, we associate "access-characteristics" with each leaf, and the algorithm's "result-characteristic" with the root.

II.3 Parse-Tree Examples:

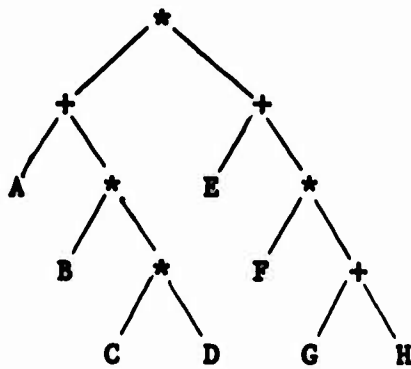
- (1) Expression: $(A+B*C*D) * (E+F*(G+H))$

Note that we have assumed a right-associative convention where ambiguity arises, as in $B*C*D$. $B*C*D$ is taken to mean $B*(C*D)$.

- (2) Parse-tree for the above expression, fully labeled. Here L stands for left, R for right.



(3) In parse-trees, we will omit arrow-heads on lines pointing downward, so that all indicated lines will be assumed to have an arrowhead on the end nearest the bottom of the page. Also, instead of explicit L or R labels, we will omit them in favor of a geometrical convention. That line drawn left-most on the page of any line directed out from a node will be implicitly labeled L. Similarly, the right-most line directed out from a node will be implicitly labeled R. These conventions allow us to draw the above parse-tree as:



Each line of this parse-tree is still labeled Left or Right, and directed, but the labeling is now implicit in the geometry of the drawing of the parse-tree.

2.4 The " n^3 " Elementary Algorithms

We present here a set of algorithms for matrix addition and multiplication based directly on the definition of these operations. Those for matrix multiplication require n^3 additions and multiplications, hence the name. We realize that a tradeoff between additions and multiplications has been achieved by Winograd which reduces the number of scalar multiplications required to $n^3/2 + n^2$. However, our techniques are not greatly affected by the new algorithms, as will be seen later, and we prefer the simpler, more familiar algorithms for most examples.

The algorithms are presented here in an abbreviated Algol notation.

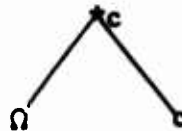
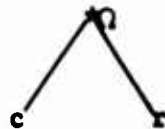
The abbreviations we use are:

Abbreviation	Algol meaning
$I \rightarrow N$	<u>for</u> $I:=1$ <u>step</u> 1 <u>until</u> N <u>do</u>
$I \xrightarrow{a} N$	<u>for</u> $I:=a$ <u>step</u> a <u>until</u> N <u>do</u>
$I \xrightarrow{-a} N$	<u>for</u> $I:=N$ <u>step</u> $-a$ <u>until</u> a <u>do</u>
$x \leftarrow e$	$x:=x+e$
<u>b</u>	<u>begin</u> (See Footnote 1.)
<u>e</u>	<u>end</u> ; (See Footnote 1.)
$x \leftarrow e$	$x:=e$

All algorithms compute $C:=A \text{ op } B$ where 'op' is either '+' or '*'. The subscript bounds are assumed 1:N in each subscript position. Additional vectors and elements used for temporary storage are introduced as needed, and are assumed to be correctly declared. Each algorithm is accompanied by a tree-like drawing, its EEPT, which abstracts certain characteristics of the algorithm. These characteristics are sufficient to determine when elementary algorithms can be combined into an "alg-tree". Their meaning, names and representations are:

1. parse-tree. Represented as a tree whose nodes are operators, or line-ends (representing variables each of whose names are suppressed.)
2. space-characteristic. Represented as a lower case letter subscript. The characteristic partially describes the order in which elements of the input matrices are accessed and in which elements of the result are computed. The letters used are chosen as follows:
 - r - "row". A row at a time is computed or accessed. The next row may be chosen arbitrarily.
 - c- "column". A column at a time is computed or accessed. The next column may be chosen arbitrarily.
 - Q- "matrix". A matrix is computed before any part is complete, or is accessed in computing one part of the result.

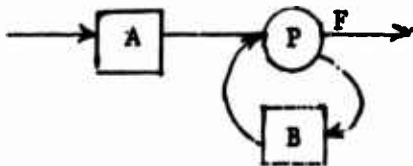
¹ Since the control structure of the algorithms is simple, we will usually delete b and e and use indentation to indicate the scope associated with matching begin-end's.

Algorithm 1: $I \rightarrow N$ $J \rightarrow N$ $C[I, J] \leftarrow 0;$ $K \rightarrow N$ $C[I, J] + \leftarrow A[I, K] * B[K, J];$ Algorithm 2: $J \rightarrow N$ $I \rightarrow N$ $C[I, J] \leftarrow 0;$ $K \rightarrow N$ $C[I, J] + \leftarrow A[I, K] * B[K, J];$ Algorithm 3: $I \rightarrow N$ $J \rightarrow N$ $C[I, J] \leftarrow 0;$ $K \rightarrow N$ $I \rightarrow N$ $J \rightarrow N$ $C[I, J] + \leftarrow A[I, K] * B[K, J];$ Algorithm 4: $I \rightarrow N$ $J \rightarrow N$ $C[I, J] \leftarrow A[I, J] + B[I, J];$ Algorithm 5: $J \rightarrow N$ $I \rightarrow N$ $C[I, J] \leftarrow A[I, J] + B[I, J];$ 


II.5 Loop Fusing

In the current section, we intend to describe a technique whereby two loops which are sequential statements of a program can sometimes fuse. The result of the fusion is a single loop which is computationally equivalent to the original sequence of two loops. This fusion requires less intermediate storage and no more operations than the original sequence. Thus, we can sometimes replace a sequence of loops with a single loop which requires less intermediate storage, and no more execution time, without changing the result of the calculation.

We will define a loop to be any program of equivalent meaning to the following flow-chart:



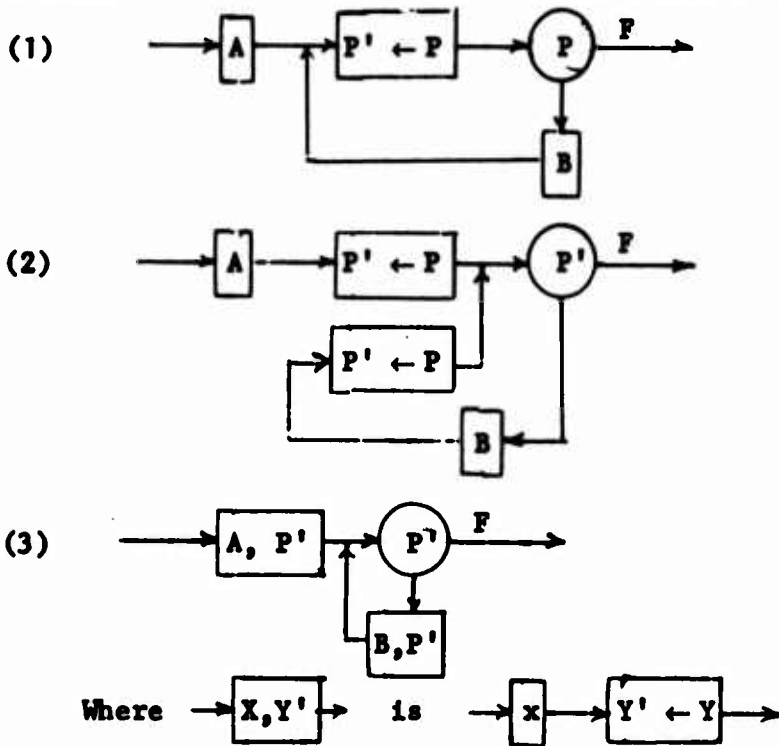
Box **A** is termed the initialization of the loop; **B** is the body of the loop; and **P** is the predicate of the loop.

If control enters any node in a flow-chart, it does so through a line directed toward the node. Further, control leaves a flow-chart node (if it leaves at all) through a line leading away from that node. We will assume that any flow-chart having exactly one entrance and one exit can be substituted for nodes drawn as square boxes: \square . Such flow-charts may include assignment statements, as well as branches, which are drawn as circles with more than one exit: . A branch may test any predicates on program variables to decide which of its exits control is to leave through; it may not, however, include assignment statements, that is it may not specify that a variable of the program be stored into.

Let U be the collection of relevant program variables. We will judge the effect of a flow-chart by its effect on these variables. That is, two flow-charts will be termed equivalent if and only if the contents of all variables in U after their executions are the same, if the contents of all variables in U before their entrances agreed.

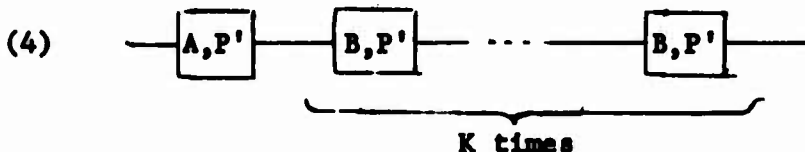
We present a series of equivalent flow-charts. Here P' is a

Boolean variable not in U , so that, in particular, it is referenced nowhere else in the flow-chart of which this loop is a part.



Flowchart 3 simplifies the predicate P so that we may assume that the decision to be made when the branch on P' is reached is pre-established, at the time the last variable in P is assigned a value.

We are attempting to emphasize the repetitious nature of a loop. In fact, if we know that P' would be set true the first K times the box $\textcircled{P'}$ was entered, then flow-chart 3 is equivalent to



Thus, a loop can be regarded as a compact abbreviation for a certain sequence of square flow-chart boxes. The "test" has no effect on the relevant program variables. Its effect is to ensure that one can create a flow-chart which can be executed a variable number of times.

We will summarize the effect of any square box on the variables in U by a set-assignment statement. Observe that the computation performed

by any box assigns certain values to some of the variables of U . The variables so changed (stored into, or assigned to are equivalent but less compact terms) are some subset of U determined by the flow-chart substituted for the box, as well as by the contents held in certain variables on entry to the box. Similarly, the values stored into these changed variables are functions of the box, and the input values. A set assignment statement describes this relation by listing in a single assignment statement the set of variables changed, the function mapping used to compute their values, and the set of variables whose values are used in the computation.

Set assignment statement example:

$$R \leftarrow f(A)$$

In the above example, R is the set of variables changed by the set-assignment statement, f is the function used to compute their values, and A is the set of variables input to the statement. Note that it may be impossible to determine the membership of each of these sets of variables without executing the flow-chart summarized by the set-assignment statement at the proper time in the program. However, we can discuss relations between these sets of variables, leaving to another problem the task of deciding if these relations are satisfied. There will be no loss of clarity in the sequel in using "assignment statement" to stand for either the usually understood assignment statement, or the set-assignment statement.

As a consequence of our introduction of the set-assignment statement, and the flow-chart equivalences sketched above, we will regard a loop as a certain sequence of set-assignment statements. We will allow the sets of variables and the functions of these statements to differ arbitrarily from statement to statement of the sequence. We will write a loop as one or more set-assignment statements, separated by semi-colons, and enclosed in square brackets:

$$[R \leftarrow f(A); S \leftarrow g(B)]$$

The sequence of statements this represents is:

$$(R)_1 \leftarrow f_1((A)_1); (S)_1 \leftarrow g_1((B)_1); (R)_2 \leftarrow f_2((A)_2); \dots$$

The center of a loop will refer to the statements enclosed in brackets in the loop's abbreviation. In the above example, the loop's center is

$$R \leftarrow f(A); S \leftarrow g(B).$$

More generally, let X be a set of variables, and S_j be some particular occurrence of a set-assignment statement, S .

Then we define $(X)_j$ to be the subset of X input to S_j

$(\underline{X})_j$ to be the subset of X stored into by S_j .

Suppose S is a statement in loop L . To refer unambiguously to an occurrence of S , J must indicate the occurrence's position in the sequence of assignments resulting from L 's iteration. J must also distinguish among the possibly several statements of the loop's center. We write $J = (L, i, k)$, where L is the name of the loop, i gives the number of the statement within the center of L (which is itself a sequence of statements), and k gives the iteration number of the loop.

We will denote the contents of a variable, or set of variables, X , just after the K th assignment statement in some sequence as $v(K, X)$. In general K is a triple, (L, i, j) giving L = loop-name, i = statement-number in loop, and j = iteration number of this occurrence, to uniquely identify the assignment-statement we mean.²

If A and B are two sets of variables, we say $v(K, A) = v(J, B)$ just when there is a one-one correspondence between A and B , and when $v(K, a) = v(J, b)$ for each pair of corresponding variables a in A and b in B .

Also, let $c(K, X)$ denote the contents of X just before the K th assignment statement.

² Triples will be identified by capital letters, and the third element of that triple will be the lower-case letter corresponding to the triple's identifier.

We will write $(A)_K \equiv (B)_J$ to mean

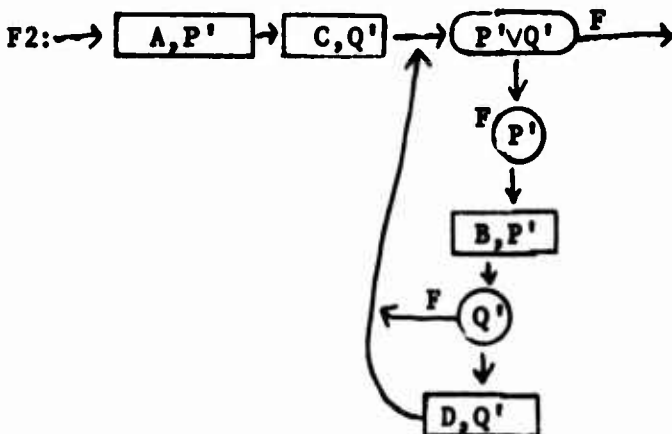
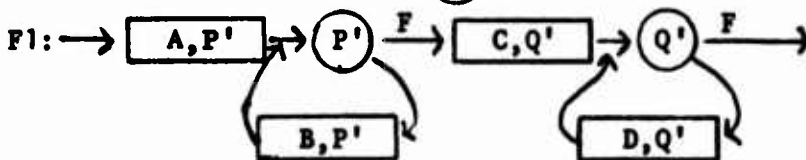
$$v(K, (A)_K) = v(J, (B)_J)$$

We say that two sequential loops, L_1 and L_2 , can fuse when a loop L_3 constructed from the components of L_1 and L_2 is computationally equivalent to the statement sequence $L_1; L_2$. In our notation, the set-assignment statement within the loop-brackets corresponds to the repeated box of the corresponding loop. In addition, various loop initialization is summarized there. We symbolize the fusion L_3 of $L_1: [R \leftarrow f(A)]$ and $L_2: [S \leftarrow g(B)]$ by

$L_3: [R \leftarrow f(A); S \leftarrow g(B)]$.

II.6 Fusion in Flow-Charts: Graphic Description

Let P' , Q' be Boolean variables not in U , so that P' occurs only in those boxes $[X, P']$ and (P') which explicitly mention it.



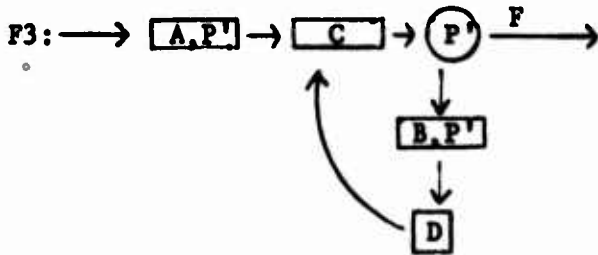
Under certain conditions, flow-chart F1 is equivalent to flow-chart F2. Flow-chart F2 is itself one loop.

Suppose $v(j, P')$ is the value P' takes on just before box (P') is entered for the j th time in flow-chart $F1$. Define $v(j, Q')$ similarly.

If

$v(j, Q') = v(j, P')$ for all j ,

then the fusion $F2$ can be simplified:



II.7 Loop Fusion Conditions

Let U be the (finite) set of all relevant program variables.

Let J be a triple (L, i, k) where L is a loop-name

i is the statement-number in L 's center, and

k is the iteration number of L .

Let $v(J, X)$ denote the contents of X just after set-assignment J .

Let $c(J, X)$ denote the contents of X just before set-assignment J .

Let $(X)[J]$, and $(X)_J$, mean the subset of X input to set-assignment J .

Let $(X)[J]$, and $(\underline{X})_J$, mean the subset of X stored into by set-assignment J .

Let $(X)_J \equiv (Y)_K$ mean $v(J, (X)_J) = v(K, (Y)_K)$.

Let $(X)_J \overset{c}{\equiv} (Y)_K$ mean $c(J, (X)_J) = c(K, (Y)_K)$.

Property of Set-Assignments:

If S_I and S_J are two set-assignments whose flow-charts are both f , then

if and only if $c(I,x) = c(J,x)$ for all $x \in (U)_I$,
then, for all f : I terminates if and only if J terminates, and

$$(U)_I = (U)_J \text{ and } (U)_I \subseteq (U)_J$$

$$\text{and } (\underline{U})_I = (\underline{U})_J \text{ and } (\underline{U})_I \equiv (\underline{U})_J$$

Theorem 1: If $(R)_j$, $(A)_j$, $(S)_j$ and $(B)_j$ are a finite collection of finite sets of variables then statements a. and b. are equivalent:

a. for all i and j ,

$$(1) (R)_j \cap (S)_i = \emptyset \text{ if } i < j \text{ and}$$

$$(2) (A)_j \cap (S)_i = \emptyset \text{ if } i < j \text{ and}$$

$$(3) (B)_j \cap (R)_i = \emptyset \text{ if } i > j$$

b. for all f and g , if there exist loops $L1$ and $L2$ whose j th set assignments are:

$$S[L1,1,j] = (R)_j \leftarrow f_j((A)_j)$$

$$S[L2,1,j] = (S)_j \leftarrow g_j((B)_j)$$

then there is a loop $L3$, constructed from $L1$ and $L2$

in the same way that $F2$ is constructed from $F1$,

and the sequence $L1;L2$ is computationally equivalent to $L3$, written ' $L1;L2 \equiv L3$ '.

An examination of conditions (1) - (3) is appropriate to illustrate the essential simplicity of the requirements. First, note that when A and B are sets of variables, " $A \cap B = \emptyset$ " means that A and B have no variables in common. We interpret a "variable" to be a unique memory cell of some processor. Then, " $A \cap B = \emptyset$ " means that the sets A and B do not share storage. If A is stored into by $S1$, and B is the set of inputs to $S2$, this means that $S1$ does not store into any variable input to $S2$, i.e., that no result produced by $S1$ is accessible to $S2$.

Condition (1), then, can be interpreted to mean that no variable

stored into by L2 is also stored into during any later iteration of L1. (2) requires that no variable input to L1 on some iteration be computed on an earlier iteration of L2. (3) states that no variable computed by L1 on one iteration can be accessed by L2 on some earlier iteration.

Control reaches a statement S just when S is the next statement to be executed.

Control is absorbed by statement S just when control reaches S, and never reaches any statement after reaching S.

Define

$P(i,j) \equiv$ "control reaches the test-statement of loop L_i for the j th time"

$[Q_1] \equiv [c((L_1,1,j),P') = \underline{\text{true}}]$

$[Q_2] \equiv [c((L_2,1,j),Q') = \underline{\text{true}}]$

$[Q_3] \equiv [c((L_3,1,j),P') = \underline{\text{true}}]$

$\vee [c((L_3,1,j),Q') = \underline{\text{true}}]$

$\neg T(i,j) \equiv P(i,j) \wedge [Q_1]_j \wedge \neg P(i,j+1)$

($\neg T(i,j)$ is true if control is "absorbed" during the j th iteration of L_i .)

Let $[R_1]_j \equiv [c((L_3,1,j),P') = \underline{\text{true}}]$

$[R_2]_j \equiv [c((L_3,1,j),Q') = \underline{\text{true}}]$

Then $T(i,j) \equiv \neg P(i,j) \vee \neg [Q_1]_j \vee P(i,j+1)$

Certain facts are easily seen from examination of flow-charts F1 and F2. These facts we call "Axioms":

Axiom 1. $P(i,j) \rightarrow P(i,j-1) \wedge [Q_1]_{j-1}, i = 1,2,3.$

Axiom 2. $[Q_1]_k \rightarrow \forall j [0 \leq j < k \rightarrow [Q_1]_j] i = 1,1,3.$

Axiom 3. $\neg [R_1]_j \rightarrow \forall k [k \geq j \wedge P(3,k) \rightarrow \neg [R_1]_k]$

Lemma 1: $P(i, j) \equiv T(i, j-1) \wedge P(i, j-1) \wedge [Q_1]_{j-1}$

Proof: By definition $T(i, j-1) \equiv \neg P(i, j-1) \vee \neg [Q_1]_{j-1} \vee P(i, j)$

from propositional calculus,

$$T(i, j-1) \wedge P(i, j-1) \equiv [\neg [Q_1]_{j-1} \vee P(i, j)] \wedge P(i, j-1),$$

$$\text{and } T(i, j-1) \wedge P(i, j-1) \wedge [Q_1]_{j-1} \equiv P(i, j) \wedge P(i, j-1) \wedge [Q_1]_{j-1} \\ \rightarrow P(i, j).$$

From the definition $P(i, j) \rightarrow T(i, j-1)$.

By Axiom 1, $P(i, j) \rightarrow P(i, j-1) \wedge [Q_1]_{j-1}$.

Lemma 2: $P(i, 0) \rightarrow$

$$P(i, j) \equiv \forall k [0 \leq k < j \rightarrow \\ T(i, k) \wedge [Q_1]_k]$$

Proof: By induction on j .

Assume $P(i, 0)$.

$$\text{Then } P(i, 1) \equiv T(i, 0) \wedge [Q_1]_0 \wedge P(i, 0)$$

$$P(i, 0) \rightarrow [P(i, 1) \equiv T(i, 0) \wedge [Q_1]_0]$$

$$\therefore P(i, 1) \equiv \forall k [0 \leq k < 1 \rightarrow T(i, k) \wedge [Q_1]_k].$$

By Lemma 1, $P(i, j) \equiv T(i, j-1) \wedge P(i, j-1) \wedge [Q_1]_{j-1}$.

from the induction hypothesis $P(i, j-1) \equiv \forall k [0 \leq k < j-1 \rightarrow T(i, k) \wedge [Q_1]_k]$

Therefore $P(i, j) \equiv T(i, j-1) \wedge [Q_1]_{j-1} \wedge \forall k [0 \leq k < j-1 \rightarrow T(i, k) \wedge [Q_1]_k]$

or $P(i, j) \equiv \forall k [0 \leq k < j \rightarrow T(i, k) \wedge [Q_1]_k]$

Lemma 3: $[Q_1]_{j-1} \wedge \forall k [0 \leq k < j \rightarrow T(i,k)] \rightarrow P(i,j)$

We know that

$\forall k [0 \leq k < j \rightarrow [[Q_1]_k \wedge T(i,k)] \rightarrow P(i,j)$, from Lemma 2.

Suppose $[Q_1]_{j-1}$. Then $\forall k [0 \leq k < j \rightarrow [Q_1]_k]$, by Axiom 2.

If $\forall k [0 \leq k < j \rightarrow T(i,k)]$, as well, then

$\forall k [0 \leq k < j \rightarrow [[Q_1]_k \wedge T(i,k)]]$,

which by Lemma 2 yields $P(i,j)$.

Therefore, $[Q_1]_{j-1} \wedge \forall k [0 \leq k < j \rightarrow T(i,k)] \rightarrow P(i,j)$.

Theorem 2: If a. of Theorem 1 holds, and

if $P(1,0)$, $i = 1,2,3$, and

$$c((L1,1,1),x) = c((L3,1,1),x) \quad \forall x \in U$$

then, for all j ,

$$\forall k \leq j \quad T(3,k) \equiv \forall k \leq j [T(1,k) \wedge T(3,k)]$$

and

$$P(3,j) \wedge T(1,j) \wedge T(2,j) \rightarrow$$

$$(X)_j = (X')_j \wedge (X)_j \equiv (X')_j, \quad X = A, B$$

$$\text{and} \quad (Y)_j = (Y')_j \wedge (Y')_j \wedge (Y)_j \equiv (Y')_j, \quad Y = R, S.$$

Proof: Suppose $P(3,j)$

$$\text{Then } \forall k [0 \leq k < j \rightarrow T(3,k) \wedge [Q_3]_k].$$

Therefore, by the induction hypothesis,

$$\forall k [0 \leq k < j \rightarrow T(1,k) \wedge T(2,k) \wedge [Q_3]_k]$$

Also, from the induction hypothesis, and the fact that

$P(3,j) \rightarrow P(3,j-1)$, we have:

$$\forall k [0 \leq k < j \rightarrow [[Q_3]_k \equiv [Q_1]_k \vee [Q_2]_k]].$$

$$\therefore \forall k [0 \leq k < j \rightarrow T(1,k) \wedge T(2,k) \wedge [[Q_1]_k \vee [Q_2]_k]].$$

From Lemma 3, and the statement above, we can deduce that

$$[Q_1]_{j-1} \rightarrow P(1,j) \quad i = 1,2$$

Because $P(3,j) \rightarrow [Q_3]_{j-1}$,

we have $[Q_1]_{j-1} \vee [Q_2]_{j-1}$,

giving $P(1,j)$ or $P(2,j)$.

Thus, if $L3$'s test is reached for the j th time, so is either $L1$'s or $L2$'s.

$$[Q_3]_j \equiv [Q_1]_j \vee [Q_2]_j, \text{ because}$$

$$\begin{aligned} \exists k < j: c((L1,1,j),P') &= v((L1,1,k),P') = v((L3,1,k1),P') \\ &= c((L3,1,j),P') \end{aligned}$$

By Lemma 3, $[Q_1]_j \rightarrow [Q_1]_{j-1}$, from Axiom 2, and

$[Q_1]_{j-1} \rightarrow P(1,j)$, $i = 1, 2$

If $\neg[Q_3]_j$, then the theorem holds, for $S[L1,k,j]$ is a

vacuous statement, which accesses and changes no variables,
and always terminates.

If $[Q_3]_j$ then

$P(1,j) \wedge [Q_1]_j$ or $P(2,j) \wedge [Q_2]_j$.

If $[Q_1]_j$, then $S[L3,1,j]$ is non-vacuous, and identical

in its set-assignment flow-chart to

$S[L1,1,j]$.

For any $x \in (A)_j$,

because $(A)_j \cap (S)_1 = \emptyset \forall i < j$,

and $(S')_1 = (S)_1 \forall i < j$ by induction hypothesis

$(A)_j \cap (S')_1 = \emptyset \forall i < j$

Therefore, $c((L1,1,j),x) \neq c((L3,1,j),x)$

only if $x \in (R')_1$ for some $i < j$.

But $(R')_1 = (R)_1$ and $(R')_1 \equiv (R)_1$ by induction hypothesis,

so x would be assigned the same value by both $S[L1,1,j]$ and $S[L3,1,j]$.

Therefore, by the properties of identical set-assignments,

$S[L3,1,j]$ terminates $\equiv S[L1,1,j]$ terminates,

and if we assume $T(1,j)$, so that $S[L1,1,j]$, then

$S[L3,1,j]$ terminates, and

$(A')_j = (A)_j \wedge (A')_j \subseteq (A)_j$ and

$(R')_j = (R)_j \wedge (R')_j \equiv (R)_j$ [line (a)]

If $\neg T(1,j)$ were assumed, so that $[Q_1]_j$ would be true,

this reasoning shows that $\neg T(3,j)$, for control would
be absorbed in the flow-chart of $S[L3,1,j]$, just as
it was in $S[L1,1,j]$.

If $\neg[Q_1]_j$, both $S[L1,1,j]$ and $S[L3,1,j]$ are vacuous,
and the theorem holds.

Regardless of whether $[Q_1]_j$ or not, if $P(3,j) \wedge T(1,j)$,

control reaches the test preceding $S[L3,2,j]$.

Again, if $\neg[Q_2]_j$, both $S[L2,1,j]$ and $S[L3,2,j]$ are
vacuous and the theorem holds.

Otherwise, $[Q_2]_j \wedge P(2,j)$.

If $b \in (B)_j$, b may have last been stored into by:

- (1) $S[L1,1,i]$, $i > 0$, or
- (2) $S[L2,1,k]$, $0 < k < j$, or
- (3) None of the above.

Case 1: $S[L1,1,i]$, $i > 0$.

Then $b \in (R)_i$, $b \notin (R)_k$ for $k > i$,

and $b \notin (S)_m$ for $m < j$.

$c((L2,1,j),b) = v((L1,1,i),b)$.

Then $b \notin (R)_k$ for $k > i$, and $b \notin (S)_m$ for $m < j$.

Hence $b \notin (R')_k$ for $j \geq k > i$, and $b \notin (S')_m$ for $m < j$.

Since $b \in (R)_i \wedge b \in (B)_j$, i must be $\leq j$,

for $(R)_i \cap (B)_j = \emptyset$ if $i > j$.

$(R')_i \equiv (R)_i$, for $i \leq j$, by induction hypothesis,

and line (a) above.

Therefore, since $i \leq j$, and $b \notin (R')_k$ for $j \geq k > i$,

and $b \notin (S')_m$ for $m < j$,

in particular $b \notin (S')_m$ for $j > m \geq i$.

Hence, $c((L3,2,j),b) = v((L3,1,i),b)$

$= v((L1,1,i),b) = c((L2,1,j),b)$.

Case 2: b last stored into by $S[L2,1,k]$, $0 < k < j$:

Then $b \in (S)_k$, $0 < k < j$, and

$$c((L2,1,j),b) = v((L2,1,k),b),$$

and $b \in (S)_k$ and $b \notin (S)_m$ for $j > m > k$.

By induction hypothesis, then,

$b \in (S')_k$ and $b \notin (S')_m$ for $j > m > k$.

$$c((L3,2,j),b) = v((L3,2,k),b) \text{ if}$$

(a) $k < j$,

and (b) $\exists m, k < m < j$ such that $b \in (S')_m$,

and (c) $\exists i, k < i \leq j$ such that $b \in (R')_i$.

(a) and (b) have been shown.

Since $(S)_k \cap (R)_i = \emptyset$ if $i > k$

$\exists i, j \geq i > k$ such that $b \in (R)_i = (R')_i$, hence (c).

Since $k < j$, by induction hypothesis

$(S)_k = (S')_k$ so

$$\begin{aligned} c((L3,2,j),b) &= v((L3,2,k),b) = v((L2,1,k),b) \\ &= c((L2,1,j),b) \end{aligned}$$

Case 3: Neither 1 nor 2 hold.

Then $b \notin (R)_i$, $0 < i$, and

$b \notin (S)_k$, $0 < k < j$.

Then, since $c((L1,1,1),b) = c((L3,1,1),b)$,

$$\text{and } c((L2,1,j),b) = c((L1,1,1),b),$$

$$\text{and } c((L3,2,j),b) = c((L3,1,1),b),$$

$$\text{then } c((L2,1,j),b) = c((L3,2,j),b).$$

Therefore, $\forall b \in (B)_j$

$$c((L2,1,j),b) = c((L3,2,j),b) \text{ and}$$

hence: $S[L2,1,j]$ terminates if and only if

$$S[L3,2,j] \text{ terminates.}$$

If we assume $\neg T(2,j)$, then $S[L2,1,j]$ and

hence $S[L3,2,j]$ do not terminate, so

$$\neg T(3,j).$$

If we assume $T(2,j)$, then

$$(B)_j = (B')_j \wedge (B)_j \subseteq (B')_j$$

$$\wedge (S)_j = (S')_j \wedge (S)_j \subseteq (S')_j$$

by the properties of set-assignments.

We have shown that

$$P(3,j) \rightarrow [T(1,j) \wedge T(2,j) \rightarrow$$

$$(X)_j = (X')_j \wedge (X)_j \subseteq (X')_j, X = A, B$$

$$(Y)_j = (Y')_j \wedge (Y)_j \subseteq (Y')_j, Y = R, S$$

$$\wedge T(3,j)]$$

and that

$$P(3,j) \rightarrow [\neg T(1,j) \vee \neg T(2,j) \rightarrow \neg T(3,j)].$$

$$\text{Therefore } P(3,j) \rightarrow [T(3,j) \equiv T(1,j) \wedge T(2,j)].$$

By the induction hypothesis, assuming $P(3,j)$,

$$\forall (k < j) T(3,k) \equiv \forall (k < j) [T(1,k) \wedge T(2,k)],$$

$$\begin{aligned} \forall (k < j) T(3,k) \wedge T(3,j) &\equiv \forall (k < j) [T(1,k) \wedge T(2,k)] \wedge \\ &\quad [T(1,j) \wedge T(2,j)], \end{aligned}$$

$$\text{or } P(3,j) \rightarrow \forall (k \leq j) T(3,k) \equiv \forall (k \leq j) [T(1,k) \wedge T(2,k)].$$

$$\text{If } \neg P(3,j), \text{ then } \exists k \ 0 \leq k < j \wedge [\neg T(3,k) \vee \neg [Q_3]_k].$$

by the induction assumption, this is equivalent to:

$\exists k \ 0 \leq k < j \wedge [\neg[T(1,k) \wedge T(2,k)] \vee [\neg[Q_1]_k \wedge \neg[Q_2]_k]]$

If $\forall k [0 \leq k < j \rightarrow T(1,k) \wedge T(2,k)]$, then, since

$\neg P(3,j) \rightarrow T(3,j)$ by its definition,

$\forall k \ 0 \leq k \leq j \ T(3,k)$ by the induction hypothesis and $T(3,j)$.

Also, $\forall k [0 \leq k < j \rightarrow T(1,k) \wedge T(2,k)]$.

Therefore, $\exists k \ 0 \leq k < j : \neg[Q_1]_k \wedge \neg[Q_2]_k$.

$\therefore \neg P(1,j) \wedge \neg P(2,j)$

From the definition, then

$T(1,j) \wedge T(2,j)$,

so $\forall (k \leq j) T(3,k) \equiv \forall (k \leq j) [T(1,k) \wedge T(2,k)]$

If $\exists k \ 0 \leq k < j \wedge \neg[T(1,k) \wedge T(2,k)]$,

by the induction hypothesis

$\exists k \ 0 \leq k < j \wedge \neg T(3,k)$

and hence

$\forall (k \leq j) T(3,k)$ and $\forall (k \leq j) [T(1,k) \wedge T(2,k)]$

are both false.

Therefore, we can assert

$\forall (k \leq j) T(3,k) \equiv \forall (k \leq j) [T(1,k) \wedge T(2,k)]$.

Proof of Theorem 1:

Part 1: a implies b.

We must show that, given identical initial conditions,

- (1) L1;L2 terminates if and only if L3 terminates
- (2) If L3 (or L1;L2) terminates, then the results computed are identical.

We have shown, in Theorem 2, that for all j,

$$\forall (k \leq j) T(3,k) \equiv \forall (k \leq j) [T(1,k) \wedge T(2,k)]$$

$$\text{or } \forall j T(3,j) \equiv \forall j [T(1,j) \wedge T(2,j)].$$

L1;L2 terminate if and only if

$$\exists j_1, j_2 : \neg[Q_1]_{j_1} \wedge \neg[Q_2]_{j_2} \wedge \forall (k \leq j_1) T(1,k) \wedge \forall (k \leq j_2) T(2,k)$$

Suppose L1;L2 terminates.

$$\text{Then } \exists j_1, j_2 : \neg[Q_1]_{j_1} \wedge \neg[Q_2]_{j_2} \wedge \forall (k \leq j_1) T(1,k) \wedge \forall (k \leq j_2) T(2,k).$$

$$\text{Let } j = \max(j_1, j_2).$$

Clearly, $\forall (k > j_1) \neg P(1,k)$, and hence $T(1,k)$.

Similarly, $\forall (k > j_2) \neg P(2,k)$, and hence $T(2,k)$.

Therefore, $\forall (k \leq j) [T(1,k) \wedge T(2,k)]$

Suppose $\neg P(3,j)$. Then $\exists (k < j) : \neg[Q_3]_k$,

$$\text{for } \forall (k \leq j) T(3,k).$$

Therefore L3 terminates

Suppose $P(3,j)$. Then $j_1 \leq j \wedge j_2 \leq j$,

$$\text{so } P(3,j_1) \wedge P(3,j_2).$$

$$\text{Hence } \neg[R_1]_{j_1} \wedge \neg[R_2]_{j_2}.$$

$$\text{Hence } \neg[R_1]_j \wedge \neg[R_2]_j.$$

Thus, $\neg[Q_3]_j$, so L3 terminates.

Suppose L3 terminates.

Then $\exists j : \neg[Q_3]_j \wedge \forall(k \leq j) T(3,k) \wedge P(3,j)$

Hence $\exists j : \neg[R_1]_j \wedge \neg[R_2]_j \wedge \forall(k \leq j) [T(1,k) \wedge T(2,k)] \wedge P(3,j)$.

$\therefore \exists j : \neg[Q_1]_j \wedge \neg[Q_2]_j \wedge \forall(k \leq j) [T(1,k) \wedge T(2,k)]$

Thus, L1;L2 terminates.

Hence L3 terminates if and only if L1;L2 terminates.

If L3, say, terminates,

$\exists j : \forall(k \leq j) [\neg[Q_3]_j \wedge T(3,k)]$

hence $\forall(k \leq j) P(3,k)$.

Then by theorem 2, $\forall k \leq j$,

$$(R)_k \equiv (R')_k$$

$$\text{and } (S)_k \equiv (S')_k$$

so the results at the time they are computed are identical.

It is conceivable however that a result computed as part of $(S)_1$ would be destroyed, in L3, by statement $S[L3,1,j]$ for some j .

This could only happen if $j > 1$ and

$$(S')_1 \cap (R')_j \neq \emptyset$$

a possibility denied by (a) of Theorem 1, which we are assuming, and Theorem 2.

If a result of $(R')_1$ were destroyed by some later iteration of L3, then

(a) If by $(R')_j$, $j > 1$, the same result would have been stored by L1.

(b) If by $(S')_j$, $j \geq 1$, the same result would have been stored by L2.

We conclude that $L1;L2 \equiv L3$.

Proof of Theorem 1, Part 2: b implies a.

We will show that $\neg a$ implies $\neg b$.

Consider any finite collection of variable-sets

$$(A)_j, (B)_j, (R)_j \text{ and } (S)_j.$$

By using additional variables as loop control elements, we can create loops L1 and L2 whose jth set-assignments are

$$(R)_j \leftarrow f_j((A)_j)$$

$$\text{and } (S)_j \leftarrow g_j((B)_j) \text{ respectively,}$$

where the control variables are designated not part of the set of "relevant" variables, U.

Now, f_j and g_j can be extremely sophisticated transformations, capable of testing their inputs for consistency. In fact, we can assume that if f_j , or g_j , ever accesses a variable whose contents differs from that supplied f_j or g_j on the jth iteration of L1 or L2, the function produces an "error reaction". Such an error reaction might take the form of a propagation of the error, by changing the contents of at least one variable

in every set $(A)_j, (B)_j, (R)_j$ and $(S)_j$. (Once some variables' value differs, f_j is not constrained to store only into variables of $(R)_j$.)

Thus, if

- (2) for any x in $(A)_j$ or $(B)_j$,
 $c((L1,1,j),x) \neq c((L3,1,j),x)$,

the result computed by L3 will differ from that computed by L1;L2. We will show that, if a1 or a2 or a3 do not hold, then some loops L1 and L2 exist for which L1;L2 does not compute the same result as L3.

Case: Suppose $(R)_j \cap (S)_i \neq \emptyset$ for some $i < j$.

Then some x exists,

$$x \in (R)_j \wedge x \in (S)_i \text{ for some } i < j.$$

After the sequence L1;L2, x 's final value was computed by $S[L2,1,1]$.

After the loop L3, x 's final value was computed by $S[L3,1,j]$. f_j and g_1 can certainly be chosen which ensure that these values differ.

Case: Suppose $(A)_j \cap (S)_i \neq \emptyset$ for some $i < j$.

Then some value input to $S[L1,1,j]$ is computed by $S[L3,2,1]$ on some iteration i before j . Thus, for some $x \in (A)_j$, $c((L1,1,j),x) \neq c((L3,1,j),x)$, and hence $L1;L2 \neq L3$.

Case: Suppose $(B)_j \cap (R)_i \neq \emptyset$ for some $i < j$.

Then there is $x \in (R)_i \wedge x \in (B)_j$ for $i > j$.

In the L1;L2 sequence,

$$c((L2,1,j),x) = v((L1,1,i),x)$$

In L3, $c((L3,2,j),x) \neq v((L3,1,1),x)$,

for iteration i of L3 follows iteration j .

Therefore, $L1;L2 \neq L3$.

II.8 Storage Savings in Loop Fusion

Once we have established the conditions under which two loops can fuse, without changing their effect, there remains the question of the storage-saving that results. We will concentrate on the intermediate storage.

Let $L1$, $L2$, and $L3$ be as before.

Define $B = \bigcup_j (B)_j$, $R = \bigcup_j (R)_j$

Recall that the size(A), where A is a set, is defined to be the number of members of A .

Then $T = B \cap R$ is the set of variables used to communicate results from $L1$ to $L2$.

We define T to be the intermediate storage set.

We will suppose that R is not input to any statement following $L1;L2$.

On the k th iteration of $L3$, certain variables must be in existence simultaneously. Those which are intermediate, that is members of T , are those which

- (1) are computed in $(R)_i$ for $i \leq k$
(since for $i > k$, they have not yet been computed and need not be present) and
- (2) are accessed by some $(B)_j$ or $(A)_j$ for $j \geq k$
(since if all such j 's are $j < k$, then the variable has already been used, and won't be needed again.)

This set is $T_k = \bigcup_{P(i,j,k,m)} [(R)_i \cap (B)_j] \cup [(R)_i \cap (B)_m \cap (A)_j]$

Where $P(i,j,k,m)$ is $i \leq k \leq j \wedge i \leq m < k$.

Here, $(R)_i \cap (A)_j$ contains those variables computed in $(R)_i$ and accessed in $(A)_j$. Some of them may not be in T . $(R)_i \cap (A)_j \cap B$ gives the set in T . For $m \geq k$ $(R)_i \cap (B) \supset (R)_i \cap (B)_m \cap (A)_j$, so these are included in the first term of T_k . This leaves $(R)_i \cap (A)_j \cap (B)_m$ for $i \leq k \wedge m < k$.

Since, in order to fuse, $(R)_i \cap (B)_m = \emptyset$ when $i \leq m$, we get

$$(R)_i \cap (A)_j \cap (B)_m \text{ for } i \leq m < k.$$

If we add two assumptions to those necessary for loop fusion, we can reduce the intermediate storage requirement still further:

$$\text{Suppose } (R)_i \cap (B)_j = \emptyset \text{ if } i \neq j$$

$$\text{and } (R)_i \cap (A)_j \cap (B)_i = \emptyset \text{ if } i \leq j$$

Then, the intermediate storage set on the k th iteration becomes

$$T'_k = (R)_k \cap (B)_k$$

and, since successive T_k 's can share the same storage,

$$\text{only } \max_k (\text{size}((R)_k \cap (B)_k))$$

variables are needed for intermediate results.

There is a second advantage to the assumption

$$(R)_i \cap (B)_j = \emptyset \text{ if } i \neq j.$$

This assumption allows the two loops to be tested separately for their fusion characteristics.

$$\text{Let } (T)_i = T \cap (B)_i$$

$$(\underline{T})_i = T \cap (R)_i$$

Theorem: $(R)_i \cap (B)_j = \emptyset$ if $i \neq j$ is equivalent to

$$[(T)_i = (\underline{T})_i \text{ and } (T)_i \cap (T)_j = \emptyset \text{ if } i \neq j.]$$

Proof: Assume $(R)_i \cap (B)_j = \emptyset$ if $i \neq j$.

$$(\underline{T})_k = T \cap (B)_k = R \cap B \cap (B)_k = R \cap (B)_k$$

$$(\underline{T})_k = T \cap (R)_k = B \cap R \cap (R)_k = B \cap (R)_k$$

But $R \cap (B)_k = (R)_k \cap (B)_k = B \cap (R)_k$ as a consequence of the hypothesis.

$$\text{Therefore } (T)_k = (\underline{T})_k.$$

If $(R)_i \cap (B)_j = \emptyset$ if $i \neq j$,

since $(T)_j \subset (B)_j$,

and $(T)_i = (\underline{T})_i \subset (R)_i$,

therefore $(T)_j \cap (T)_i \subset (B)_j \cap (R)_i \subset \emptyset$ if $i \neq j$.

Assume $(T)_i = (\underline{T})_i$ and $(T)_i \cap (T)_j = \emptyset$ if $i \neq j$.

Then $(T)_j = T \cap (B)_j = R \cap B \cap (B)_j = R \cap (B)_j$

and $(T)_i = (\underline{T})_i = T \cap (R)_i = B \cap (R)_i$

When $(T)_j \cap (T)_i = \emptyset$

$\emptyset = [R \cap (B)_j] \cap [B \cap (R)_i] = (B)_j \cap (R)_i$

Since $(B)_j \subset B$ and $(R)_i \subset R$.

Hence $(B)_j \cap (R)_i = \emptyset$ if $i \neq j$.

II.9 Summary of Loop Fusion Conditions

We have described a set of conditions which guarantee that two loops, L1 and L2 can fuse to yield one loop, computationally equivalent to the sequence L1; L2. A slight strengthening of these conditions results in a fusion which uses less intermediate storage.

Let L1; L2 be

L1: $[R \leftarrow f(A)]$

L2: $[S \leftarrow g(B)]$

Then there is a loop

L3: $[R' \leftarrow f(A'); S' \leftarrow g(B')]$

constructed by fusing the flowcharts of L1 and L2 in such a way that statements of L1 alternate with statements of L2.

If

$$(1) \quad (R)_j \cap (S)_1 = \emptyset \text{ if } 1 < j$$

$$(2) \quad (R)_j \cap (B)_1 = \emptyset \text{ if } 1 > j$$

$$(3) \quad (S)_j \cap (A)_1 = \emptyset \text{ if } 1 > j$$

all hold, then

L1; L2 is computationally equivalent to L3.

If initial conditions agree, so that

$$c((L1, 1, 1), v) = c((L3, 1, 1), v) \text{ for all } v \text{ in } A \cup B,$$

then finally for all j

$$(X)_j = (X')_j, \quad X \in \{A, B, R, S\}$$

$$(Y)_j \stackrel{c}{=} (Y')_j, \quad Y \in \{A, B\}$$

$$(Z)_j \equiv (Z')_j, \quad Z \in \{R, S\}$$

If the storage conditions

$$(1') \quad (R)_j \cap (S)_1 = \emptyset \text{ if } 1 < j \quad (\text{same as (1)})$$

$$(2') \quad (R)_j \cap (B)_1 = \emptyset \text{ if } 1 \neq j$$

$$(3') \quad (S)_j \cap (A)_1 = \emptyset \text{ if } 1 > j \quad (\text{same as (3)})$$

$$(4') \quad (R)_1 \cap (B)_1 \cap (A)_j = \emptyset \text{ if } 1 \leq j$$

all hold, then the storage need for communication from L1 to L2,

$$T = \bigcup_{L,j} [(B)_i \cap (R)_j]$$

becomes

$$T' = \max_k [(B')_k \cap (R')_k] \text{ in } L3$$

(Here "max" of a series of sets selects that set which contains most members.)

II.10 Algol and Flow-Chart Language

We have introduced a flow-chart language to clarify the meaning of certain programs-parts. However, we consider any programming-language construct having the same meaning as one of these terms to be the "same" as that term. In particular, we will often refer to restricted forms of certain programming constructs in Algol.

Algol of course contains "assignment statements". Suitably restricted versions of these, which store into only one relevant program variable, agree with our (unstated previously) concept of an "assignment statement".

Furthermore, certain Algol FOR-statements agree with our definition of "loop". An Algol FOR statement whose FOR-clause specifies no GOTO's and whose body, *S*, specifies no GOTO's which lead outside *S* satisfies our definition of loop. (A GOTO may be specified implicitly, as part of a procedure body which is called in the text, or explicitly in the text. Both are excluded.)

II.11 Algorithm Fusion (parallel connection)

We wish to describe how two algorithms, which occur in sequence, can fuse to reduce the storage needed for communication from the first to the second. We will base our analysis on the properties which allow loops to fuse. Here, we must isolate the loops in the two algorithms which can fuse to save storage. If these loops are separated by one or more statements, they must be rearranged, preserving computational equivalence, to make them adjacent.

Let *S*₁ and *S*₂ be adjacent simple algorithms. Let *R* be a set of variables which is the result-set of *S*₁, and an input-set to *S*₂. Under certain circumstances, *S*₁ and *S*₂ can combine, or parallel connect, allowing *R*, the storage used to communicate between *S*₁ and *S*₂, to be reduced in size.

Conditions:

- (1) The result computed by *S*₁ into *R* is input to no statement sequence other than *S*₂;
- (2) There is a loop, *L*₁, of *S*₁ which encloses all statements of *S*₁ which store into *R*;

- (3) There is a loop, L2, of S2 which encloses all statements of S2 which access the values stored in R by L1.

Let $(R)_j$ be the set of elements of R which are stored into during the jth iteration of L1. Let $(R)_j$ be the set of elements of R which are input during the jth iteration of L2.

Further conditions:

- (4) $(R)_j = (R)_j$ for all j such that there is a jth iteration of L1 and of L2.
- (5) $(R)_j \cap (R)_i = \emptyset$ if $i \neq j$
- (6) S1's change-set is disjoint from each input to S1, and from the change-set, S, of S2.
- (7) No variable stored into by S2 is accessed by S1.

Theorem: If conditions 1-7 above are met, there exists a single algorithm S3 computationally equivalent to the sequence S1;S2. Furthermore, R may be replaced by a smaller set V of variables, where $\text{size}(V) = \max_j [\text{size}((R)_j)]$.

Proof: Conditions (1), (4)-(7) imply that L1 and L2 may fuse if they are adjacent. If L1 and L2 fuse, then these same conditions allow R to be reduced in size to V, where $\text{size}(V)$ is as above. We must show that L1 and L2 can be made to be adjacent.

In the statement sequence S1;S2, suppose there are statements S0 following L1 in S1. Since, by (2), L1 is the only statement in S1 which stores into R, S0 does not store into R. Since R is the only input to S2 which is computed by S1, and since no variable accessed by S1 is stored into by S2, S0 can follow S2 without affecting the computation. Now, suppose there are statements S4 of S2, which precede L2 in S2. Since all accesses of R lie in L2, S4 cannot access R. Since S2 does not store into any variable accessed by S1,

neither does S4. Further, S4 cannot depend on any result computed by S1, for it does not depend on R's contents, and, since R is S1's result-set, it does not depend on any other variable's contents computed by S1. Therefore, S4 can be moved to precede S1 without changing the computational effect. The result of these moves is a statement sequence in which no statements intervene between L1 and L2. L1 and L2 can then fuse, and the result follows.

II.12 Matrix Operation Algorithms (MOA's)

The results of the previous section can be applied to the specific algorithms which are used in computing matrix arithmetic expressions.

A matrix operation algorithm (MOA) is a simple algorithm which computes an associated matrix assignment statement. Syntactically, a matrix assignment statement is written " $V \leftarrow E$ ", where V is a matrix identifier, and E is a matrix arithmetic expression. Semantically, this matrix assignment statement commands the replacement of the contents of the array named V with the value of the matrix arithmetic expression, E. The value of E is computed from the contents its variables hold immediately before the statement is executed. The result-set of the MOA is the array V; the input-set is the union of the arrays whose names appear in E. A copy of an MOA is a systematic substitution instance of the MOA. A substitution instance of an MOA results from substituting new arrays for the arrays referenced by the MOA. Suppose an MOA, B, computes the matrix assignment statement " $V \leftarrow E$ ". A substitution, S, which changes array X to S(X), is systematic if and only if, for all arrays R whose name occurs in E, if $\text{name}(R) = \text{name}(V)$ then $\text{name}(S(R)) = \text{name}(S(V))$ and if $\text{name}(R) \neq \text{name}(V)$ then $\text{name}(S(R)) \neq \text{name}(S(V))$. The same substitution may then be applied to the MOA's matrix assignment statement, to yield the new MOA's associated matrix assignment statement.

For example,

$A \leftarrow B * C$, $A \leftarrow B * B$, $A \leftarrow D * E$ are all copies of $X \leftarrow Y * Z$. However, $A \leftarrow A * C$ is not a copy of $X \leftarrow Y * Z$, for since X is the variable stored into, and $X \neq Y$, the new names for these variables, S(X) and S(Y) must not agree.

A substitution which is not systematic cannot in general be applied to an MOA without changing the MOA's assignment statement radically. For example, suppose the (not systematic) substitution

A replaced by A
C replaced by A
B replaced by B

is applied to Algorithm 1 of the n^3 algorithms.

The algorithm becomes:

I \rightarrow N
J \rightarrow N
A[I,J] \leftarrow 0
K \rightarrow N
A[I,J] \leftarrow A[I,K] * B[K,J]

This algorithm most definitely fails to compute

A \leftarrow A * B,

for some element of each row of A is set to zero before it can be accessed. Precisely what it computes is not clear, but it certainly does not compute the value of A * B, and store this value in A, as the assignment statement A \leftarrow A * B requires.

An algorithm to compute any matrix arithmetic expression can be constructed from copies of the MOA's in a suitably chosen basic set of MOA's. An example of a basic set of MOA's is any set containing MOA's to compute the matrix assignment statements

A \leftarrow B + C and A \leftarrow B * C

Suppose we are given a set of MOA's containing MOA's that compute

A \leftarrow B + C and A \leftarrow B * C.

We can use sequences of copies of these MOA's to compute the value of any matrix arithmetic expression, E.

Proof: by induction on the number of operators in the expression E .

If E is a matrix identifier, V ,

then the value of E is defined to be the contents of V . But the null sequence of MOA-copies computes just this, in V .

If E is of the form V_1 op V_2 , where V_1 and V_2 are matrix identifiers,

and op is + or *, then the single MOA-copy which computes $Z \leftarrow V_1$ op V_2 computes the value of E into Z . Z must be chosen to differ in name from both V_1 and V_2 .

If E is of the form E_1 op E_2 , where E_1 and E_2 are expressions

containing at least one operator, and op is + or *, then E_1 and E_2 can each be computed, into any prescribed arrays X_1 and X_2 , by sequences of MOA-copies C_1 and C_2 . Choose X_1 different from any array occurring in C_2 , and write the sequence " $C_1; C_2; \text{op } X_2$ ", where Z differs from X_1 and from X_2 . This sequence computes the value of E into Z .

Let us call this technique COMPl.

The sequence of MOA's produced by COMPl for a given expression E requires several arrays to hold values of subexpressions of E . The semantics of the expression prevents these intermediate values from ever being input to another statement in the program. In fact, the result produced by each MOA in the sequence, if it is not the final MOA's result, is input to only one other MOA. The sequence also has the property that, if the expression E contains no common subexpressions, no subexpression of E is ever computed more than once. Thus, this technique produces a minimum-connection-time MOA to compute any matrix arithmetic expression.

Refinements of COMPl for translating matrix arithmetic expressions into sequences of MOA-copies can be derived, which reduce the

number of arrays used in producing the expression's value. Notice that the technique given requires that certain arrays must be chosen to be different from others used in certain MDA's. Thus notice the constraint on X_1 imposed in COMP1. This prevents one algorithm's result-set from being stored into during the course of another, before that result-set is accessed by the algorithm which must receive its contents. However, it forces arrays not needed to hold the value of the expression to be used to carry these intermediate results. Some of these intermediate arrays can be eliminated.

At least two techniques for refining COMP1 exist. One can, using only the two given MDA-assignments, reduce the number of intermediate arrays used, by re-ordering the sequence of execution of the component MDA's. This technique works by changing the statements over which a given intermediate array must remain intact, and hence, be a distinct, non-usable storage area. The maximum number of such distinct intermediate arrays occurring in an expression's translation is the number of arrays needed to compute that expression. This number can be minimized. Alternatively, one can develop new algorithms, capable of computing expressions with more operators. If these algorithms themselves each need so little temporary storage (say, a factor of N less than an array) that it can be discounted, the "larger" the expressions which can be computed using only one array to hold results. Both techniques can be employed together, as well.

From certain sequences of two adjacent simple algorithms, we can reduce the storage needed for their communication to the amount needed during one of their outer loop iterations. Suppose that we have translated some expression, using COMP1. The resulting sequence of MDA's has many pairs of communicating adjacent MDA's, each pair using an array for communication. This suggests the use of algorithm parallel connection, to produce a collection of algorithms for the basic set which compute expressions having more than one operator, and yet need a negligible amount of intermediate storage.

II.13 Shapes, the "Valences" of an MOA

A convenient summary of the combining properties of MOA's can be devised. First, note that the variables used for communication between two algorithms are, in the case of MOA's, organized into arrays. Furthermore, COMPI shows that we can freely choose these arrays. In fact, we could choose a distinct array for each pair of communicating MOA's in the sequence COMPI produces. Note also, that once an array of the sequence is accessed by an MOA, its contents is not needed by any other MOA. These considerations, checked against the requirements which allow two algorithms to parallel-connect, rapidly reduce the potentially unsatisfied conditions.

Of the conditions allowing two algorithms to parallel-connect, only a few are possibly unsatisfied in an appropriate sequence of MOA's. Since any communicating storage in such a sequence takes the form of an array, it seems natural to investigate the remaining requirements by characterizing each array used to hold inputs or results of an MOA.

Let us define, for each array X of an MOA which occurs in a loop L of the MOA, the element-sets of that array. $(X)_{L,j}$ will represent the subset of variables of array X input to L during L 's j th iteration. Similarly, $(\underline{X})_{L,j}$ will represent the subset of X stored into during L 's j th iteration. For any array X , these sets represent a subset of X 's variables selected by a certain subset of the possible subscript combinations by which elements of X are selected. Let the collection of subscripts in $(X)_{L,j}$ be $[X]_{L,j}$ and similarly let $[\underline{X}]_{L,j}$ select $(\underline{X})_{L,j}$. Now define the shape associated with an array X to be the sequence of subscript-sets $[X]_{L,j}$ or $[\underline{X}]_{L,j}$ according as X is input to or stored into during a single outermost loop L of the algorithm. If X occurs in more than one outermost loop of the algorithm, let its shape be defined to be Ω . Furthermore, if $[X]_{L,j} \cap [X]_{L,i} \neq \emptyset$ (or $[\underline{X}]_{L,j} \cap [\underline{X}]_{L,i} \neq \emptyset$) for some $i \neq j$, let X 's shape be Ω .

The shape Ω is assigned to an array to prevent fusion at this array with other MOA's. Assignment of Ω to an array in effect demands the presence of an entire array to hold values during a computation. When-

ever it is difficult to assign a "combinable" (non- Ω) shape to some array, Ω may be assigned, without altering the correctness of the algorithm resulting from parallel-connection at that array.

The concept of "shape" is our promised useful summary of an algorithm's parallel-connection property. In some sense, it corresponds to a chemist's "valence": we state that MOA B can parallel connect to MOA C at input X of C if the shape of the result-array of B and the shape of X in C match. Two shapes match if neither are Ω , and if they are equal, element-set by element-set, for the first K element-sets of the sequences where K is defined as the largest number j such that iteration j of either loop exists. We call the shape associated with an input array X of an MOA C, the access-characteristic of X in C. The shape associated with the result-array of an algorithm C is called the result-characteristic of C. The algorithm resulting from the parallel connection is called the fusion of A to input X of B.

We can simplify shape-comparisons considerably by assigning descriptive names to the most commonly occurring shapes. The table below lists these names, together with a one-letter abbreviation for each, defining them in terms of the subscripts their Jth element-set is selected by:

r - row. All $[J, i]$ such that $1 \leq i \leq N$

c - column. All $[i, J]$ such that $1 \leq i \leq N$

Ω - Ω . No subscript set corresponds. Given to "unfuseable" arrays.

Theorem: If the result-characteristic of MOA A matches the access characteristic of input X of MOA B, and if A's change-set is disjoint from each of A's input-sets, and if B's result array is disjoint from X, then there exists an algorithm C such that

- (1) C computes a matrix assignment statement derived by substituting the expression of a substitution of A's associated matrix assignment statement for each occurrence of X in B's matrix assignment statement.

- (2) C is constructed as the sequence $S(A); S(B)$, with $S(A)$ parallel connected to input X of B. $S(A)$ is a systematic substitution of A, such that A's result becomes X in $S(A)$. X in B remains X in $S(B)$, but B's change-set is chosen differently from any variable A.

Proof: The sequence $S(A); S(B)$ computes in $S(B)$'s result-set, the required matrix assignment statement. We will take this as the result of C. We can therefore assume that no value computed into variables not in the result-set of B is input to any later statement in the program, by the definition of result-set. It remains to show that $S(A); S(B)$, a sequence of two adjacent simple algorithms, can fuse.

We observe that:

- (1) the result computed by $S(A)$ into X is input to no statement-sequence other than $S(B)$, by our definition of the result-set of the sequence.
- (2) The result-shape of A, and hence of $S(A)$ does not equal Ω . Therefore, there is a single outermost loop $L1$ of $S(A)$ which encloses all statements of $S(A)$ which store into X.
- (3) Similarly, the access-shape of X in B, and hence in $S(B)$, does not equal Ω . Hence there is a single outermost loop $L2$ of $S(B)$ which encloses all statements which access X in $S(B)$.

Let $(X)_j$ stand for the subset of variables of X stored into during $L1$'s j th iteration, and $(X)_j$ be the subset of X input during the j th iteration of $L2$.

- (4) The result-shape of A equals the access-shape of X in B, because the non- Ω shapes match. Hence, the

subscript-sets $[X]_j = [\underline{X}]_j$ for all $j < K$. But applying identical subscripts to the same array, X , selects identical variables. Hence $(X)_j = (\underline{X})_j$ for all $j < K$. Then $(X)_j = (\underline{X})_j$ for all j such that there is a j th iteration in $L1$ and $L2$.

- (5) $(X)_j \cap (\underline{X})_i = \emptyset$ if $i \neq j$, for if it weren't so, the result-characteristic of A would be Ω .
- (6) $S(A)$'s change-set is disjoint from each of $S(A)$'s input-sets by assumption. $S(A)$'s change-set is disjoint from $S(B)$'s change-set by construction of the substitutions.
- (7) $S(B)$'s change-set is disjoint from any variable in $S(A)$, so no variable stored into by $S(B)$ can be accessed by $S(A)$.
- (8) No variable stored into by $S(A)$ is input to any statement other than $S(B)$, by our definition of the sequences result-set to include only variables in $S(B)$'s result-set.

Thus, all the hypotheses of the parallel-connection theorem are satisfied.

II.14 Explicit Rule for Developing Shapes for Arrays used in Algol Programs

In general, a shape cannot be calculated for each array occurring in an MOA. The difficulty arises because the element-sets which constitute the shape may depend on values computed during the algorithm. In general, because we cannot predict these values, we cannot decide the membership of the element-sets.

The Algol programs we use as illustrations, however, all reference element-sets in a particularly simple way. Their loops are thoroughly predicable FOR-statements, for which the value of the FOR-statement's index at the start of the loop's j th iteration is easily calculable. These indices are the only variables appearing in array subscripts.

Furthermore, no conditional statements occur to skip statements, leaving variables unreferenced, even though their subscript-combination apparently appears. For these loops, we can, and do, calculate shapes.

II.15 Parallel Connection Algorithm

Suppose we are given two Algol simple algorithms A and B such that A's result is input to B. Suppose that A's result-characteristic and B's access-characteristic match. Then A and B may be parallel-connected. Here, we make explicit how.

First, set down A immediately preceding B. A's result-characteristic is a non-Q shape. Therefore, there exists an outermost loop, L1, of A, enclosing all statements of A which change A's result. Similarly, an outermost loop of B, L2, exists, enclosing all accesses of A's result. Move any intervening statements out from between L1 and L2, moving those statements of A after B and those of B before A. Fuse L1 and L2, by deleting L2's controlling for-clause, and the now-adjacent end-begin pair which enclosed it. Replace the for-clause with statements to ensure that L2's index is stepped in exactly the way it was stepped by L2's for-clause. Now, substitute an intermediate variable name for A's result throughout the combined algorithms.

This technique of fusing two Algol loops fails to account for possible "conflicts" of indexes. Semantically, an Algol FOR-loop's controlled variable, or index, takes on an "undefined" value after the FOR-list is exhausted. We can take this to mean that this value may not be input to any statement in the program. Therefore, we may substitute a new variable for the FOR-statement's index, without changing the meaning of that FOR-statement. We use this property to avoid such conflicts.

Suppose I1 is L1's index, and L1 and L2 are to fuse. If, in L2, I1 is stored into, we say a "conflict on I1" exists. To avoid it, we simply substitute for I1 a variable not occurring in L1 or L2.

We have been somewhat vague in describing how we insure that L2's

index is stepped exactly the same way L2's FOR-clause stepped it. In general, this can be accomplished by substituting an appropriate sequence of conditional statements and assignments. The Algol report [6] suggests such sequences for each FOR-list element-type. By "expanding" the FOR-clause into the simpler statements it abbreviates, and then redetermining the statement the loop's exit is to reach, L2's index can be stepped.

In certain loop-fusions, a simpler approach can be used. Suppose that L1 and L2 are to fuse, and that each iterates the same number of times. Suppose also that I1 is L1's index and I2 is L2's.

Suppose that the first statement in the Algol text of each loop's body is given the number "1", and the last in loop L1's text is numbered K1.

If there is a function F such that $F(v((L1, K1, j), I1)) = c((L2, 1, j), I2)$ for all j then statements to compute I2 from I1's current value directly may replace I2's iterative computation. These statements may be placed just before the first statement of L2's body in the fusion. If no statement of L2 stores into I2 (the usual case), then $F(I1)$ may replace each occurrence of I2. [In many cases, such an F exists--the identity function.]

Example 1

a: $I \rightarrow N$ $A: r$
 $J \rightarrow N$ $B: \Omega$
 $C[I,J] \leftarrow 0$ $\underline{C}: r$
 $K \rightarrow N$
 $C[I,J] + \leftarrow A[I,K] * B[K,J]$

b: $I \rightarrow N$ $D: r$
 $J \rightarrow N$ $E: \Omega$
 $A[I,J] \leftarrow 0$ $\underline{A}: r$
 $K \rightarrow N$
 $A[I,J] + \leftarrow D[I,K] * E[K,J]$

We can reduce A from a 2-array to a 1 array:

b: $I \rightarrow N$
 $J \rightarrow N$
 $A[I,J] \leftarrow 0$
 $K \rightarrow N$
 $A[I,J] + \leftarrow D[I,K] * E[K,J]$

Fusion begun:
 delete the second
for clause.

a: $(I \rightarrow N)$
 $J \rightarrow N$
 $C[I,J] \leftarrow 0$
 $K \rightarrow N$
 $C[I,J] + \leftarrow A[I,K] * B[K,J]$

Reduce the size of A to a row by assigning the Ith element of the Jth row to U[I]; i.e., substitute U[x] for A[y,x] wherever A occurs:

```

I → N
  J → N
    U[J] ← 0
    K → N
      U[J] + ← D[I,K] * E[K,J]
    J → N
      C[I,J] ← 0
      K → N
        C[I,J] + ← U[K] * B[K,J]

```

Space characteristics: D: r
 E: Ω
 B: Ω
 C: r

Example 2:

```

b. [ J → N          D:  $\Omega$ 
    [ I → N          E: c
      [ A[I,J] ← 0   A: c
        [ K → N
          [ A[I,J] + ← D[I,K] * E[K,J]
a. [ I → N          A: c
    [ J → N          B: r
      [ C[I,J] ← 0   C:  $\Omega$ 
        K → N
          J → N
            I → N
              C[I,J] + ← A[I,K] * B[K,J]

```

J is the index of b, K that of a. Simple replacement conditions are met. An F(J) which can replace K throughout a, is 'J'. A name conflict arises, so choose J1 for both J in b and K in a. Let $U[x] = A[x,y]$.

and upon fusing, we get:

I → N	D: Ω
J → N	E: c
C[I,J] ← 0	B: r
J1 → N	<u>C</u> : Ω
I → N	
U[I] ← 0	
K → N	
U[I] + ← D[I,K] * E[K,J1]	
J → N	
I → N	
C[I,J] + ← U[I] * B[J1,J]	

II.16 Matrix Elementary Algorithms

Let us define the matrix elementary algorithm of size k (k-MEA's) to be a subset of the MOA's satisfying certain constraints:

- (1) There is a single loop in each algorithm, called its main loop, whose input-set includes each variable input to the algorithm, and which stores into each variable of the algorithm's result-set.
- (2) All arrays of the algorithm assigned non- Ω shape are k-arrays, i.e., have sizes of N^{**k} .³
- (3) Each non- Ω shape associated with an array of the algorithm consists of $1*N$ equal-sized element sets. Here, j may differ depending on the shape, but may not depend on N . Hence, each element-set of a non- Ω shape of a k-MEA has size proportional to $N^{**}(k-1)$
- (4) Each k-MEA uses no more than $L*N^{**}(k-1)$ intermediate variables. Here, L does not depend on N .

Some consequences of our definition of k-MEA follow:

Theorem 1: If two k-MEA's fuse, the result is a k-MEA.

Theorem 2: The inputs to a k-MEA must all be present simultaneously at some instant in time.

Theorem 3: If two k-MEA's fuse, any input to either except the result of the first becomes an input to the fusion.

³We will use the FORTRAN notation $**$ for exponentiation and $*$ for multiplication

Proof of Theorem 1: Suppose A and B are k-MEA's, and C is the fusion of A to input X of B. Then C is a simple algorithm, since the initial step of the fusion process merely lists the statements of A before those of B. The sequence of two simple algorithms is itself a simple algorithm. We must show that C is a k-MEA. We first need to show that C contains one loop, which stores into each variable of the result of C, and accesses each variable of each input-set of C.

- (1) The result-set of C is defined to be the result set of a substitution of B. B, and hence $S(B)$ are MEA's, and therefore there is a loop, L2, which stores into each element of $S(B)$'s result-set. Furthermore, L2 accesses each variable input to $S(B)$. In particular, X is an input to L2. But X's access characteristic is not Ω , since A can fuse to X of B. Therefore, X must be input only to L2, and hence L2 is the loop of $S(B)$ which fuses with some loop of $S(A)$. Also, A's result-characteristic is not Ω , and A is an MEA. Similarly, there is a loop L1 of $S(A)$ to which each input of A is input, and which must be the loop which fuses with L2. The fused loop, L, has, as inputs, all inputs to $S(A)$ and to $S(B)$ (except X). The inputs to L thus include all the inputs to C. Furthermore, L stores into each variable in C's result-set, for it stores into B's result-set. Therefore, L satisfies the definition of a k-MEA's main loop.
- (2) All arrays of the original k-MEA's were k-arrays. Hence all arrays of the fusion, a subset of the arrays of the original k-MEA's, are k-arrays.

- (3) Each shape associated with an array of B or of A other than X is still the same, since loop-fusion does not change the element-sets of inputs or result-sets. Therefore, each array has a shape consisting of $j \cdot N$ equal-sized element-sets, since it did so in the original k-MEA's.
- (4) Let A and B be the k-MEA's which fuse to form C. Then A used no more than $j_1 \cdot N^{k-1}$ intermediate variables, B no more than $j_2 \cdot N^{k-1}$. C uses, at most, all intermediate variables of A and of B, plus those variables used to hold X, which is intermediate in C. But the fusion reduces the number of variables needed for X to one element-set of X, or $j_3 \cdot N^{k-1}$ variables, by property (3) of a k-MEA, and the fact that, since A parallel connects to B at X, X must have a non- Ω shape in A and in B. Thus, C uses at most $(j_1 + j_2 + j_3) \cdot N^{k-1}$ intermediate variables, where j_i do not depend on N.

Proof of Theorem 2: The inputs to a k-MEA must all be present simultaneously just before the algorithm's execution, for they are all inputs to the same loop of the k-MEA. By definition of "input", the contents of each variable just before the loop is executed is accessed by this loop. But then all variables to the loop must have been in existence simultaneously just before the loop executed. These inputs are precisely the inputs to the k-MEA, by definition of k-MEA.

Proof of Theorem 3: The inputs to the main loop of the first algorithm are clearly inputs to the main loop of the fusion, because if they were accessed by the first's main loop, they are now accessed by the fusion's main loop. Similar reasoning holds for each variable input to the second's main loop. However, some of those variables are now not in existence before the fusion, since in the sequence they were results of the first k-MEA. Hence, all input variables to either MEA except the result of the first k-MEA become variables of the fusion.

II.17 Canonical k-MEA's

A canonical k-MEA is a k-MEA which computes a matrix assignment statement satisfying:

No identifier appearing in the expression (right-side) of the assignment agrees in name with the left-side variable of the statement.

Equivalently, the result-set of a canonical k-MEA is disjoint from each of its input-sets.

Theorem: If the result-characteristic of a canonical k-MEA A matches the access-characteristic of input X of canonical k-MEA B, then A may parallel-connect to input X of B to form a fusion C which is itself a canonical k-MEA.

Proof: A and B are MOA's satisfying the hypothesis of the theorem of Section 13. Therefore, they may fuse to form an MOA. This MOA is a k-MEA, by Theorem 1 of Section 16. The fusion k-MEA computes a matrix assignment statement whose expression results from substituting a systematic substitution instance of A's expression for each occurrence of X in B. B is canonical, so that no input-array of B has the same name as B's result-array. The substitution instance of A can be so chosen

to arrange that no input array of A is given the same name as B's result-array. But then C is canonical.

Chapter III

In this chapter, we apply the results of Chapter II to matrix arithmetic expressions. Our goal is an algorithm which compiles these expressions into canonical 2-MEA's, choosing a compilation which uses fewest 2-arrays. We will, throughout this chapter, use "array" to abbreviate 2-array, and "MEA" to abbreviate 2-MEA.

III.1 Elementary Expression Parse-Trees (EEPT's)

In order to study the possible compilations of an expression into MEA's, it is convenient to examine parse-trees, both of the expression, and of the available primitive MEA's. The significance of a parse-tree in expression compilation stems from the fact that, for the expressions we deal with, a parse-tree is a data-flow diagram for the expression. That is, if x and y are nodes of a parse-tree, and y is x 's father, then x cannot be evaluated after y , for x 's result is an input to the computation which yields y 's result. [x can, however, be calculated in pieces, at the same time parts of y are being calculated.] The parse-tree, or its generalization the data-flow diagram gives a partial ordering of the times of calculation of the expressions rooted at each parse-tree node. It is therefore extremely useful in displaying all possible valid linear orderings of those computations.

Each canonical MEA can be abbreviated by a tree diagram. The structure of this diagram is a parse-tree of the expression the MEA computes. The names of the input arrays, and the result are suppressed, since systematic renaming of these variables yield computationally equivalent algorithms. However, we associate with each leaf of the tree, and with the root, shapes--the access-characteristic and result-characteristic of the algorithm abbreviated. The diagram, called an elementary expression parse-tree (EEPT) conveniently summarizes the fusion and computation behavior of the algorithm it abbreviates.

Using EEPT's as building blocks, we can devise a technique for

producing a large collection of canonical MEA's. Suppose EEPT-1's result-characteristic matches the access-characteristic of a leaf L of EEPT-2. Construct the tree diagram which results from attaching EEPT-1 to L. This tree is the parse-tree of an expression composed of the expressions of EEPT-1 and EEPT-2. Furthermore, this new expression abbreviates an algorithm which is itself a canonical MEA.

To construct the canonical MEA abbreviated by the joining of two EEPT's, we can proceed as follows:

1. Write EEPT-1's algorithm, A, immediately before EEPT-2's algorithm, B.
2. Rename the matrices used in A and in B so that A's result-matrix agrees with the input-matrix of B associated with the leaf L of EEPT-2 whose access-characteristic matched the result-characteristic of EEPT-1, and so that the name of B's result-array does not agree with the name of any array input to A, or to B.
3. Fuse A and B. This fusion can be accomplished, because A's result-characteristic agrees with B's access-characteristic at the common communicating array.

III.2 Alg-Tree Definitions

Let us assume that we are given:

1. An expression's parse-tree, E.
2. A collection of EEPT's, e_1, \dots, e_n .

Each EEPT represents an algorithm-class, any one of whose members can compute the subexpression described by the EEPT's parse-tree. We wish to assign to each intermediate node of E a method drawn from these algorithms for computing that node from its descendants. Thus, we want a correspondence set up between each intermediate node of E and a unique intermediate node of a unique e_i .

Definition: We say that an elementary expression tree "e" matches E at z if and only if

1. For each node α of e there exists one and only one node x of E , its corresponding node.
2. If α, B are nodes of e joined by a line from α to B labeled i , then so are their corresponding nodes in E .
3. If α is an intermediate node of e , then its operator-symbol matches the operator-symbol of its corresponding node x in E .
4. The root-node of e corresponds to z .

If e matches E at x , then the set of nodes of E corresponding to leaves of e are termed the fringe-set of e at x . We will often identify the fringe-set by listing the set of its node-names. Associated with each line incident on a member of a fringe-set of e at x is the corresponding line of e . These terminal lines of e have an access-shape characteristic which is thereby associated with lines of E . The access-shape associated with line L is termed the line-shape(L). If only one line L is incident on a node x , we define leaf-shape(x) = line-shape(L). Also, the root of e is associated with a shape attribute, called the root-shape of e .

Definition of an alg-tree:

An alg-tree of a node x in E is an assignment of elementary expression parse trees [EEPT] e_i to certain nodes of E . The assignment satisfies the following construction property:

1. An EEPT which matches E at x is an alg-tree of x in E .
2. If T is an alg-tree of x in E , then if e is an EEPT "parallel attachable" to T at some node y of E , then T extended to the nodes matched by e is also an alg-tree of x in E .

The nodes in an alg-tree T are the nodes of E assigned EEPT-nodes by T . The root of an alg-tree T is that node in T having no ancestor node in T . The fringe-set of an alg-tree T is that set of nodes which

have no immediate descendants in T . The connection-set of an alg-tree T consists of that set of nodes of T which corresponds to the root of some EEPT assigned by T to E .

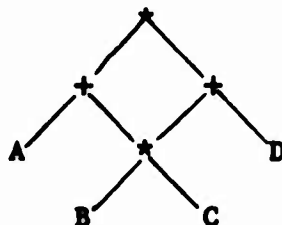
Definition of parallel-attachable:

An EEPT e is parallel-attachable to an alg-tree, T , in E at a node y if and only if:

1. y is in the fringe-set of T
2. e matches E at y
3. The line-shape, s_1 , assigned by T to each line i incident on y equals e 's root-shape.
4. Every line incident on y comes from a node in T .

The above definitions allow E the possibility of being a re-entrant tree, one in which a given subtree may have several ancestors. This corresponds to a generalization of a parse-tree to a data-flow diagram for expressions with common subexpression.

Example: An expression with common subexpressions, such as $(A + B * C) * (B * C + D)$ would be represented as:



Suppose e is an EEPT which abbreviates an algorithm \underline{e} . Let e match E at x , so that x_1, \dots, x_k are the nodes of E matching leaves of e . Let the result of a node x be the value of the expression whose parse-tree is rooted at x . If the results of each x_i are stored in arrays X_i , executing a suitable version of \underline{e} produces the result of x . We say that the EEPT can compute the result of the node its root matches.

Associated with each alg-tree A is a canonical MEA \underline{A} whose parse-tree matches the portion of E in A . If the results of each node in the fringe-set of A is available to \underline{A} , \underline{A} can compute the result of the

root of A . \underline{A} is constructed from the algorithms abbreviated by the EEPT's assigned in A . One way of constructing such an associated algorithm proceeds in parallel with the construction of \underline{A} :

- (1) Suppose A is an EEPT e which matches E at x . Then let \underline{A} be e 's canonical MEA.
- (2) Suppose A is an alg-tree T of x , parallel-attached to an EEPT e at y . Then construct \underline{A} from \underline{T} , T 's canonical MEA, and \underline{e} , e 's canonical MEA, as follows:
 1. Systematically substitute a new matrix name, Z for \underline{e} 's result-matrix, and for the input \underline{y} of \underline{T} corresponding to y . Here, Z is a name not occurring in \underline{T} or \underline{e} .
 2. Write the substituted \underline{e} before the substituted \underline{T}
 3. Parallel connect \underline{e} to \underline{T} , yielding a canonical MEA.

The inputs to the resulting algorithm include all inputs to \underline{T} and to \underline{e} .

Steps 1 and 2 can always be accomplished. Step 3 can also be accomplished, for if A assigns e to match y , the root-shape of e must agree with the leaf-shape of y in T . But the root-shape of e is the result-characteristic of \underline{e} and the leaf-shape of y in T is the access-characteristic of \underline{y} in \underline{T} . The result of \underline{y} is needed only as an input to \underline{T} , since all lines in E incident on y are in T . Therefore, Z as computed by \underline{e} need be input only to \underline{T} . Therefore, once \underline{e} and \underline{T} have been placed in sequence, they can parallel-connect.

To show that the algorithm produced indeed computes the proper result, observe that if \underline{A} is \underline{e} , we have already demonstrated the result. Suppose then that \underline{A} is the fusion of \underline{e} with input \underline{y} of \underline{T} . \underline{A} is then computationally equivalent to $\underline{e};\underline{T}$. But immediately after \underline{e} is executed in this sequence, all of \underline{T} 's inputs are available. Since \underline{T} is T 's associated algorithm, \underline{T} computes the result of the root of A .

III.3 Major Properties of Alg-Trees

Alg-trees are interesting primarily because they explore the mechanism by which canonical MEA's may be constructed to fit a given expression-part. This growth-mechanism is itself important in growing alg-trees efficiently. Thus, the first of the three major alg-tree properties concerns only the growth-mechanism--the EEPT's, not their algorithms. The two additional properties we discuss are primarily properties of the algorithms which can be grown in parallel with alg-trees, the "associated alg-tree algorithms" (ATA).

Property 1

Let $G(S,x)$ be the set of all alg-trees of x whose root's shape is S . Let $H(y,S,x)$ be the set of alg-trees of y which include x in their connection-set, and such that the root-shape of the EEPT they assign to x is S .

Then: For all $U \in G(S,x)$, there is $T \in H(y,S,x)$ such that U is a sub-alg-tree of T . For all $T \in H(y,S,x)$, there is $U \in G(S,x)$ such that U is a sub-alg-tree of T .

In other words, the set of subtrees of members of $H(y,S,x)$ rooted at x and with root-shape S equals $G(S,x)$, the set of all alg-trees of x with root-shape S .

We will use this result in growing the alg-trees rooted at y from $G(S,x)$. Every alg-tree of x with root-shape S can be "extended upward" by parallel-attaching it to any EEPT A which matches some ancestor-node y of x such that x lies in A 's fringe-set, and such that x 's leaf-shape in A is S . These extensions ultimately create all alg-trees rooted at y .

Proof: Each member T of $H(y,S,x)$ contains a sub-alg-tree (which may be null, if x is in T 's fringe-set) which is rooted at x with root-shape S . Clearly this is a member of $G(S,x)$. Furthermore, this sub-alg-tree may be replaced by any member of $G(S,x)$, each replacement yielding an alg-tree rooted at y , a member of $H(y,S,x)$.

The remaining major properties of alg-trees concern, in reality, AATA's.

Property 2 Each AATA is a canonical MEA.

Two consequences of Property 2 are:

- a. Each AATA requires only $k \cdot N$ intermediate variables, since it is a 2-MEA. Thus, the storage internal to an AATA does not enter the leading term of the polynomial in N which counts the number of memory cells (variables) needed by the program.
- b. If the AATA of an alg-tree T of x is used to compute x 's result, then the result of every node in the fringe-set of T must be computed, and be present, before T 's AATA can execute.

Proof: The fringe-set of T lists the limits of the fusion making up T 's AATA. The results of these nodes are inputs to T 's AATA, and must therefore be present simultaneously just before the AATA's execution, since each AATA is an MEA.

Property 3 A non-canonical MEA can be produced from certain canonical MEA's which are AATA's of alg-trees. By a "non-canonical" MEA, we mean one whose result-set may be assigned the same variables as one of the MEA's input-sets. For example, from the canonical MEA to compute

$$A \leftarrow B * C$$

we can produce non-canonical MEA's which compute

$$A \leftarrow B * C \quad \text{and} \quad A \leftarrow B * A.$$

Such non-canonical MEA's allow us to re-use intermediate variables immediately, thus avoiding the need to provide an additional distinct array to hold the result of an MEA.

Theorem: If the result-characteristic of an MEA, A , matches the access-characteristic of an input I of A , then the result set of A may be chosen to be I 's input-set. The MEA which results from this transformation requires one "move" operation for each element of I 's input set, in addition to the operations needed by A .

Proof: Suppose A 's result-characteristic matches the access-characteristic of an input I of A . Then, each element-set of the result-set computed during a single iteration j of A 's main loop equals (in subscript sets) the set of elements of I accessed during iteration j . Set A 's result-array equal to I 's input array. Include instructions in A 's main loop to perform

$$U \leftarrow (I)_j$$

just before the body of A 's main loop. Within the body of A 's main loop, refer to the copy of $(I)_j$ in U whenever the contents of a variable in $(I)_j$ is accessed. A 's result, now $(\underline{I})_j$, is computed into the same variables in $(I)_j$ by this resulting algorithm. Because the shapes of A 's result and I "match", neither is \emptyset , and since $(I)_j \cap (I)_i = \emptyset$ if $i \neq j$, no variables in $(I)_j$ will be accessed on any iteration other than the j th; Because $(\underline{I})_j = (I)_j$, no variable is stored into during iteration j which is not in $(I)_j$, and hence copied into U .

III.4 Result-Array and Fringe-Set Array Storage Overlap

At the end of the preceding section we presented a demonstration that, if the root-shape of an alg-tree matches a leaf-shape, and if an intermediate array was assigned to that leaf, then that same array can be used to hold the result. Our demonstration involved copying the set $(R)_j$ into intermediate storage just before computing $(\underline{R})_j$. It appears that this "move" or copy operation is only a bookkeeping convenience, and that at no cost in intermediate storage it can be eliminated.

An example follows:

Suppose we wish to compute

$$X \leftarrow X * B$$

where X and B are 2-arrays, and X is intermediate. We can compute X (final) with only one $1 * N$ array of storage and no copy as follows:

Instead of X , allocate Y , an $(N+1) * N$ array, and compute X (initially) in the first N rows of Y , ie.

$$X[I,J] = Y[I,J]$$

This leaves $Y[N+1, J]$ empty.

Now substitute into algorithm 1 (see "the n^3 basic algorithms") as follows:

for $C[I,J]$ substitute $Y[I+1, J]$
 for $A[I,J]$ substitute $Y[I,J]$

Also, reverse the sequence of values computed by the outer loop.

The result is:

$$\text{Case: } X[I,J] = Y[I,J] \quad 1 \leq I, J \leq N$$

$$I \rightarrow N$$

$$J \rightarrow N$$

$$Y[I+1,J] \leftarrow 0;$$

$$K \rightarrow N$$

$$Y[I+1,J] + \leftarrow Y[I,K] * B[K,J];$$

X (final) is computed by this algorithm in the last N rows of Y .

This suggests another case, differing in X 's initial position in Y . Here we again use algorithm 1, this time substituting

$$Y[I,J] \text{ for } C[I,J]$$

$$Y[I+1,J] \text{ for } A[I,J]$$

and running the outer loop in normal order.

Case: $Y[I+1,J] = X[I,J]$

$I \rightarrow N$

$J \rightarrow N$

$Y[I,J] \leftarrow 0$

$K \rightarrow N$

$Y[I,J] + \leftarrow Y[I+1,K] * C[K,J]$

Virtually the same construction may be used in the outer loop of any canonical MEA whose result-characteristic matches one of its access-characteristics. The direction of the main loop is determined by the "position" of the input which is to be overstored in its larger intermediate array.

Similar constructions can be used to overstore an input whose access-shape is "c". However, the array in which the input to such an algorithm is stored must be allocated somewhat differently: $N * (N+1)$ rather than $(N+1) * N$, allowing an extra column. Consideration of the $N * (N+1)$ arrays (extra-column arrays), and c-access algorithms with input matrices stored in extra-row arrays indicates a proliferation of cases. One simple solution would simply allocate each intermediate array to be $(N+1) * (N+1)$. This results in only four possible locations for the (1,1) element of a matrix stored in such an array, T:

$[T[1,1], T[1,2], T[2,1] \text{ and } T[2,2]]$

The appropriate r-access and c-access algorithms can easily be calculated in each of these cases.

A similar construction applies to any shape of an algorithm such that:

- (1) An additional element-set of variables may be allocated as an extension of the sequence of element-sets which make up the shape, and
- (2) The element-sets of the shape may be computed (and accessed) in reverse order without changing the result of the algorithm.

When these conditions are satisfied for some shape S, then any canonical MEA whose result-shape equals S may overstore any of the MEA's input-arrays whose access-shape also equals S.

III.5 Calculating Intermediate 2-Array Requirements of an Alg-Tree

One of the properties of alg-trees suggests that, in some cases, a "large" algorithm with many inputs to the same outer loop may require more space than the equivalent "smaller" (fewer input) algorithms. This is the "parallelism" inherent in parallel attachment, namely, that all inputs to an alg-tree must exist simultaneously. Given the set of inputs to an algorithm, the results at each node of the algorithm's fringe-set, we must determine the number of arrays needed to evaluate all members of the fringe-set and which must be simultaneously present.

Suppose we are given a fringe-set, each node i in it requiring $n(i)$ arrays for its computation. In computing the entire fringe-set, one intermediate array is needed to hold each node's result unless its $n(i) = 0$, for in this case, the member is a leaf of the expression, and already exists as a program variable. If node i is computed j th among the fringe-set members, then apparently we need at least

$$n(i) + j - 1 - k(j)$$

arrays to compute it. Here $k(j)$ is the number of nodes L such that $n(L) = 0$ which are to be computed before node i .

Let the number of arrays required to compute the fringe-set S be $N(S)$.

Then $N(S) \geq n(i) + j - 1 - k(j)$ for each i

Let S be a fringe-set, a set of nodes numbered arbitrarily 0 to m .

Let J be any permutation on the integers in $[0, m]$.

With each permutation J , we can associate an order for computing the nodes of S . Namely, if these nodes are numbered $0, \dots, m$, then the i th node we compute is numbered J_i . Thus, we can define

$$N(J, S) = \max_i (n(J_i) + i - K(J, i))$$

where $K(J, i)$ is the number of elements J_k in S such that $k \leq i$ and $n(J_k) = 0$. Then the following result holds:

Theorem: Any permutation J for which

$$N(J_i) \geq n(J_{i+1}) \quad 0 \leq i < m$$

minimizes $N(J, S)$, and for that permutation

$$N(S) = \max_{\substack{0 \leq i \leq m \\ n(J_i) > 0}} \{n(J_i) + 1\}$$

Proof: Let $N(J, S) = \max_i \{n(J_i) + 1 - K(J, i)\}$, where $K(J, i)$ is the number of k 's such that $n(J_k) = 0$ and $k \leq i$.

We have $N(S) = \min_J N(J, S)$.

In a following lemma, we prove that:

If for some i , $n(J_i) < n(J_{i+1})$, then there is a permutation J' such that

$N(J, S) \geq N(J', S)$, where J' is defined:

$$J'_k = J_k \quad \text{if } k < i \text{ or } k > i + 1$$

$$J'_i = J_{i+1}$$

$$J'_{i+1} = J_i$$

Therefore, if there is an i such that

$$n(J_i) < n(J_{i+1})$$

then there is a permutation J' which interchanges J_i and J_{i+1}

and so that

$$n(J', S) \leq N(J, S)$$

By a sequence of such interchanges, we can arrive at J'' , a permutation in which

$$(1) \quad n(J''_i) \geq n(J''_{i+1})$$

even if we started with a permutation J for which $N(J, S)$ took on its minimum value. Furthermore, $N(J'', S) \leq N(J, S)$.

All permutations J'' satisfying (1) produce identical values of $N(J'', S)$.

Therefore, $N(S) = N(J'', S)$ for all J'' satisfying (1).

Lemma: If for some i $n(J_i) < n(J_{i+1})$, then there is a permutation J' such that $N(J, S) \geq N(J', S)$, where J' is defined:

$$J'_k = J_k \quad \text{if } k < i \text{ or } k > i + 1$$

$$J'_i = J_{i+1}$$

$$J'_{i+1} = J_i$$

Proof: Let $T_i = n(J_i) + i - K(J, i)$.

$$\text{Then } N(J, S) = \max_i T_i$$

Similarly, let $T'_i = n(J'_i) + i - K(J', i)$

$$\text{so } N(J', S) = \max_i T'_i$$

Suppose $n(J_i) < n(J_{i+1})$

Then $n(J_{i+1}) > 0$, and either $n(J_i) = 0$ or $n(J_i) > 0$.

Case: $n(J_i) = 0$.

$$\text{Then } K(J, i) = K(J, i+1) = K(J', i+1)$$

$$\text{while } K(J', i) = K(J, i) - 1.$$

$$T_{i+1} = n(J_{i+1}) + i + 1 + K(J, i+1)$$

$$\begin{aligned} T'_i &= n(J_{i+1}) + i + K(J', i) \\ &= n(J_{i+1}) + i + K(J, i+1) - 1 \end{aligned}$$

$$\text{Therefore, } T'_i < T_{i+1}$$

Also,

$$\begin{aligned} T'_{i+1} &= n(J_i) + i + 1 + K(J', i+1) \\ &= n(J_i) + i + 1 + K(J, i+1). \end{aligned}$$

Therefore, $T'_{i+1} < T_{i+1}$, when

$$n(J_i) < n(J_{i+1}).$$

Case: $0 < n(J_1) < n(J_{i+1})$.

$$\text{Then } K(J,1) = K(J,i+1) = K(J',1) = K(J',i+1)$$

$$T_{i+1} = n(J_{i+1}) + 1 + 1 + K(J,i+1)$$

$$\begin{aligned} T'_1 &= n(J_{i+1}) + 1 + K(J',1) \\ &= n(J_{i+1}) + 1 + K(J,i+1). \end{aligned}$$

Therefore, $T'_1 < T_{i+1}$.

$$\begin{aligned} T'_{i+1} &= n(J_1) + 1 + 1 + K(J',i+1) \\ &= n(J_1) + 1 + 1 + K(J,i+1) \end{aligned}$$

But $n(J_1) < n(J_{i+1})$ by hypothesis,

so $T'_{i+1} < T_{i+1}$.

For all $k < i$ or $k > i+1$,

$$T'_k = T_k$$

Therefore, we have shown that for all x , there is a k such that

$$T_k \geq T'_x$$

In particular, for that x which maximizes T'_x , there is a k such that

$$T_k \geq T'_x = N(J',S)$$

Since $N(J,S) \geq T_k$,

$$N(J,S) \geq N(J',S).$$

Thus far we have shown how to calculate the number of arrays needed both to compute each input to an algorithm, and to hold those inputs simultaneously just before the algorithm executes. We have not, however, shown how many are needed to complete the algorithm's execution, including computation of its result. This number, $n(S)$, depends on whether the algorithm's result may overstore a fringe-set array, or not. We claim:

$n(S) = N(S)$ if the result may occupy an array holding one of the algorithm's inputs.

$= N(S \cup I)$ otherwise.

Here, I is a node-name distinct from all the names of nodes of S , and such that $n(I) = 1$.

When the result may not overstore an array of the alg-tree's fringe-set, we must account for the possibility that the members of the fringe-set each require more than one array in their computation, but of course only one to hold their result. Let the non-zero-members-of- S be the set of members x of S such that $n(x) > 0$. Then

$n(S) = \max (N(S), \text{number of non-zero-members-of-} S + 1)$.

But $n(S) = N(S \cup I)$ computes precisely this.

We can say then, that

$n(S) = N(S)$ if: (1) the root-shape of the alg-tree whose fringe-set is S matches the leaf-shape of some node x in S , and
(2) x is not a leaf of E .⁴

and $n(S) = N(S \cup I)$ otherwise.

⁴The second condition is required, for we cannot change the value of any variable of the expression in computing that expression.

III.6 The Leaves-In Algorithm

Property 1 of an alg-tree enables us to avoid some of the redundant alg-tree growing that simple application of our minimization method would require. We can record, for each shape S at each node, all the distinct alg-trees with shape S . We can then generate these sets for any node x , given that they have been generated and recorded for all descendants of x .

To generate all members of $G(S, x)$, where x is not a leaf of E :

- (1) Choose an EEPT e , whose root-shape equals S , which matches E at x . Let the nodes of E corresponding (in the match) to leaves of e be y_1 , certain descendant nodes of x . Let the leaf-shape of each y_1 be s_1 .
- (2) For every i , choose a member of $G(s_1, y_1)$. Let the set of alg-trees so chosen be $\{F_1\}$.
- (3) Extend the assignment B of e to E made in (1) to include all assignments to F_1 . B' , the extension of B , is defined to assign to each node x assigned an EEPT node \underline{x} by an F_1 or by B that same EEPT node \underline{x} .
- (4) Each distinct choice in step (1) or (2) generates a new alg-tree in (3). Repeat until all choices are made.
- (5) Repeat steps 1-4 for each distinct shape S .
- (6) Steps 1-5 create a set of alg-trees B_1 rooted at x . Provide for the possibility that x may be in the fringe-set of some alg-trees rooted at ancestors y of x , by computing $n(x)$, an integer giving the minimum number of arrays needed to compute x regardless of which B_1 is used. A "null alg-tree", which has only one input, x , having $n(x)$ as its array requirement, must then be added to $G(S, x)$ for each S .

When x is a leaf of E , compute $G(S, x)$ = the null alg-tree, with a single member in its fringe-set, x . $n(x) = 0$; for no

intermediate arrays are needed in computing a leaf of E . Leaves of E are assumed to be computed before any subexpression of E is evaluated.

Theorem: If no node in the parse-tree E has more than one line incident on it, we can verify that this algorithm computes all, and only, the members of $G(S,x)$ for each S .

Proof: To show that each object produced by step (3) is a member of $G(S,x)$ for the S chosen in (1), we must show that each such object is an alg-tree rooted at x , with root-shape S . In step (3) we extend an assignment of an EEPT e to x to other nodes of E . We must show that the resulting assignment is an alg-tree. Clearly, by the construction rule for alg-trees, e 's assignment to x is an alg-tree, T . e 's i th line-shape, s_i , is assigned a line incident on node y_i of E by T . F_i is an alg-tree rooted at y_i whose root-shape is s_i , by step (2). This means that the EEPT f_i whose root is assigned y_i by F_i has root-shape s_i . Also, f_i matches E at y_i . Since no node of E has more than one line incident on it, y_i 's only incident line has line-shape s_i . All lines incident on y_i have shape s_i , and lie in T . Therefore f_i is parallel-attachable to T at y_i , and hence the extension of T to include f_i is an alg-tree rooted at x with root-shape S . Succeeding extensions can be made to include all EEPT's used in f_i 's construction. Similar reasoning shows that T may be extended to each F_i . Hence, each object produced by step (3) is a member of $G(S,x)$.

To see that all members of $G(S,x)$ are produced by the leaves-in algorithm, assume the contrary. Then there is a $T \in G(S,x)$ not produced for any choices of e 's and F_i 's made in steps (1) and (2). T differs from each assignment produced by the leaves-in algorithm in at least one node. T must, however, assign to x an EEPT which matches E at x and has root-shape S . Since e matches E at x in only one way, each node of E assigned by T to nodes of e matching

E at x is assigned that same node by one of the choices of (1). In particular, the leaves of e are assigned the same nodes y_i and shapes s_i in T as in some choice of EEPT made by (1). Thus, $T \in H(x, s_i, y_i)$ for each i . Therefore for each i there is an $F_i \in G(s_i, y_i)$ which is a sub-*alg-tree* of T . Some choice made by (2) selects precisely these F_i 's for every i . Then T cannot differ from this selection on any node in F_i . But T assigns only nodes of E in an F_i , or which match nodes of e . Thus, there is one selection of choices in steps (1) and (2) from which T cannot differ.

In performing step (6) of the leaves-in algorithm, we find that we must compute the minimum number of arrays needed to compute the result of a node x . This in turn requires an evaluation of the number of arrays needed for the execution of each *alg-tree* rooted at x . These numbers depend on the array requirements of the nodes in each *alg-tree*'s fringe-set. We will speak of "the value of" an *alg-tree* A , or a node x , when we mean the minimum number of arrays needed for the execution of A to yield x 's result, or for the computation of x 's result by the *alg-tree* rooted at x whose value is least.

If we were to represent *alg-trees* as EEPT-node identifiers attached to certain nodes of E , linked together in some way, each time we needed to compute an *alg-tree*'s value, that *alg-tree*'s fringe-set would have to be obtained. A better representation for *alg-trees* avoids much of this computation. We could represent *alg-trees* by their fringe-sets. In order for this change of representation to save computation, we would like to show that operations analogous to the steps of the leaves-in algorithm can be performed on fringe-sets, to yield new fringe-set representations directly. But only step (3), the step which produces a new *alg-tree* depends on *alg-tree* representations. If each of the F_i input to step (3) were represented as *alg-tree* fringe-sets, step (3) could produce the fringe-set of the extension F of e to the F_i by set-uniting all the F_i fringe-sets. [Each node in the fringe-set of F must have been a fringe-set node of some F_i . Similarly, each node in the fringe-

set of some F_1 is in the fringe-set of F .] The change of representation is thus desirable.

A still more desirable representation of alg-trees presents itself. In computing the value of each alg-tree, we apply the function n to the alg-tree's fringe-set, S , yielding $n(S)$. $n(S)$ ultimately requires the evaluation of $N(S)$ (or $N(S \cup I)$). Recall that a fringe-set S is a certain collection of nodes in a graph. In evaluating $N(S)$, the names of the nodes in S are irrelevant. $N(S)$ requires only $n(S_1)$, the values of the nodes S_1 in S , for its computation. This suggests that each fringe-set, represented as a list of integer node-names $\{y_1, \dots, y_m\}$ be associated with the fv-set $(n(y_1), \dots, n(y_m))$. The fv (fringe-value) set we will represent as a string of integers separated by spaces. Each integer $n(y_1)$ is the value of some node y_1 in the associated fringe-set and is the number of arrays needed in computing y by that algorithm which uses fewest arrays.

We will extend the functions $N(S)$ and $n(S)$ to apply to fv-sets. If S' is the fv-set associated with S , with S'_i being the integers in S' ,

$$N(S') = \max_{\substack{1 \leq i \\ S'_i > 0}} (S'_i + 1 - 1)$$

$$\text{where } S'_i \geq S'_{i+1} \text{ for all } S'_i, S'_{i+1} \text{ in } S'$$

Thus, $N(S') = N(S)$ (cf. the definition of $N(S)$).

Each alg-tree can be represented by its fv-set during the leaves-in algorithm. We must describe once more how step (3) of the leaves-in algorithm can be modified to accomodate the new representation. So long as there is no node y of E with more than one line incident on y (an assumption which the leaves-in algorithm requires in any case), no two fringe-sets united by the modified step (3) have nodes in common. For two fringe-sets united by step (3) must include descendants of two distinct nodes y_1 and y_j only. Furthermore, y_1 is neither a descendant nor ancestor of y_j , for both are members of the fringe-set of an alg-tree, and nodes of a fringe-set of an alg-tree have no descendants in that alg-tree, and hence in that fringe-set. The lack of nodes in the

tree with more than one incident line implies that the set of descendants of y_i is disjoint from the set of descendants of y_j . Hence, the fringe-sets united by step (3) include no nodes in common.

The fact that two fringe-sets united by the modified step (3) have no nodes in common suggests a simple extension of fringe-set set-uniting to the fv-sets of those fringe-sets. We can define the "join" of two fv-sets $A = (A_1, \dots, A_m)$ and $B = (B_1, \dots, B_r)$ to be $(A_1, \dots, A_m, B_1, \dots, B_r)$. That is, the join of A and B, written $A \cup B$, is a set of integers consisting of every integer appearing in A or in B. The number of integers in the join is the sum of the number of integers in A and the number in B. Step (3) of the leaves-in algorithm can be modified to read:

- (3') Join the fv-set representation of all the F_i to produce the fv-set representation of a new alg-tree.

Further information will be associated with each fv-set computed, in objects called the "tags" of each fv-set. Tags explicitly represent alg-trees, which, although not needed during the leaves-in algorithm, nonetheless must be recoverable, for the alg-trees constitute the desired output of the procedure. Alg-trees will be explicitly represented by linking each fv-set F produced with the fv-sets F_i joined in step (3') to form F. Tags will also hold additional information associated with each fv-set, notably the root-EEPT number, and the fringe-shape-set. Except for the fringe-shape-sets, none of the information in tags is essential in computing fv-sets.

The root-EEPT of an alg-tree A is the EEPT assigned by the alg-tree to the node at which A is rooted. Each fv-set when generated is placed in one of the sets $G(S, x)$. Such sets are termed shape-sets. At each node one shape-set is produced for each distinct EEPT root-shape. Shape-sets represent information about the fv-sets they contain, information which is needed by the leaves-in algorithm.

Fringe-shape-sets enable step (6) to compute $n(S)$ from $N(S)$, where is an fv-set. Recall that

$n(S) = N(S)$ if some node x in the fringe-set S satisfies:

- (1) the leaf-shape of x matches the root-shape of S 's alg-tree; and
- (2) x is not a leaf of E .

$n(S) = N(S \cup I)$ otherwise.

Fringe-shape-sets record, associated with each fv-set, sufficient information to allow these two cases to be distinguished. A fringe-shape-set $F(S)$, where S is its associated fv-set, is a subset of all possible shape names. Presence of a shape name t in $F(S)$ records the presence, in S 's fringe-set, of some node x satisfying (2), whose leaf-shape equals t . Suppose C is an fv-set belonging to shape-set $G(S, x)$. Then, if $F(C)$ is C 's fringe-set, the proper evaluation of $n(C)$ can be determined:

$n(C) = N(C)$ if $S \in F(C)$,

$n(C) = N(C \cup I)$ otherwise.

Here $C \cup I$ is an fv-set formed by joining an extra integer I to C . The introduction of a fictitious node I to achieve the proper value is no longer necessary.

Fringe-shape-sets must be computed along with fv-sets. Thus, again using $F(C)$ to denote the fringe-shape-set of fv-set C , if G_i are the fv-sets to be joined in step 3', to form fv-set G , compute as well

$$F(G) = F(\bigcup_i G_i) = \bigcup_i F(G_i)$$

Here $F(A) \cup F(B)$ is just the set union of $F(A)$ and $F(B)$. Clearly, if shape-name t occurs in, say $F(A)$, it means that some node I of A 's fringe-set has leaf-shape t . In the fusion, I will retain leaf-shape t . Hence, t should occur in $F(A \cup B)$. Furthermore, if t occurs in neither $F(A)$ nor $F(B)$, no member of the fringe-set of $A \cup B$ will have leaf-shape t (for no nodes other than those in the fringe-set of A or the fringe-set of B enter the fringe-set of $A \cup B$), and hence t must not be in $F(A \cup B)$. Occasionally, we will need to represent fringe-shape-sets explicitly, as in examples. We will represent (on paper) a fringe-

shape-set $F(C)$ by listing, immediately after fv-set C , the shape-abbreviations contained in $F(C)$. Thus, if

C is an fv-set containing the integers 1,1,2,3,
and $F(C)$ contains the shape r,c ,

we represent C and $F(C)$ together as:

(1 1 2 3) r c

To complete the computation of fringe-shape-sets, we must produce a representation for the "null alg-tree", added in Step 6 to each shape-set $G(S,x)$. The only nodes of E assigned by the null alg-tree are in its fringe-set. This fringe-set consists of a single node, x . The fv-set representation of this alg-tree would therefore be $FN = (n(x))$. We can compute $Z = \min_i n(C_i)$, where each C_i is an fv-set generated in shape-set $G(S,x)$ for some S (including $S = \Omega$). Each C_i represents a method for computing x . That C_j among the C_i which requires fewest arrays for x 's computation is chosen. The number of arrays C_j requires is $n(C_j)$ --hence the computation of Z gives $n(x)$. The links of FN , which will represent the alg-tree to be used to compute x , should be copies of the links of C_j . We argue that the fringe-shape-set of the copy of $(n(x))$ added to $G(S,x)$ should be S . For FN in reality represents the computation of x into an intermediate array. When FN is added to $G(S,x)$, any fusion with this copy of FN will, by step (2) of the leaves-in algorithm, access this array by shape S . Hence, the fringe-set of FN contains a node, x , which is not a leaf of E , and whose leaf-shape is S . Therefore $F(FN) = S$, since x is the only node in the fringe-set of FN .

One more refinement of our algorithm can be introduced. Step (2) of the leaves-in algorithm requires us to select a member of $S(s_i, y_i)$ for each leaf i of EEPT e . But what if $s_i = \Omega$? Such a shape is not defined to "match" another of the same name. We resolve this difficulty simply. We arrange that $G(s_i, y_i)$ contain only fv-set $(n(y_i))$. This fv-set represents the null alg-tree, with which an alg-tree whose leaf-shape is Ω at y_i may "fuse". Furthermore, since in reality y_i will belong to the fringe-set of the fusion, treating the fv-set $(n(y_i))$

consistently as an fv-set introduces the integer $n(y_1)$ into each fv-set F for which y_1 belongs to F 's fringe-set, and to no others. We accomplish this by adding Step (7) to the leaves-in algorithm.

- (7) Replace $G(Q, x)$ with the null alg-tree rooted at x . The representation of this alg-tree is the fv-set

$$FN = (n(x))$$

$F(FN)$ is empty

The handling of the leaves L of E is straightforward. Clearly only the null alg-tree matches a leaf of E . Hence, each shape-set $G(S, L)$ contains only $FN = (0)$, for $n(L) = 0$, because no intermediate arrays are required to compute a leaf. Furthermore, because L is a leaf of E , $F(FN)$ is empty in each shape-set, for the only node in the fringe-set of FN is L which is a leaf of E . Hence, no shapes occur in the fringe-shape-set of any copy of FN .

From now on we will discuss tagged fv-sets and the alg-trees they represent interchangeably. Each of the terms here defined for alg-trees can be extended to apply to the tagged fv-sets representing alg-trees. Thus, we will speak of the root-shape of fv-set G , meaning the root-shape of the alg-tree G' whose fv-set is G , and which the tags of G represent, etc.

III.7 Effort Estimates Motivating Search Reduction

It is worthwhile at this point to make an estimate of the number of alg-trees we must consider. The time we spend in optimizing an expression is likely to be directly related to this number.

One of our early formulations of this problem suggested that a very large number of alg-trees would have to be considered. We supposed that a maximal alg-tree rooted at a node x was given. The leaves of such an alg-tree either coincide with leaves of E , or have null leaf-shape, so that no EEPT can be parallel attached to them. We then observed that each "pruning" of a branch of this alg-tree resulted in a

new alg-tree. We can calculate the number of such alg-trees derivable by such branch paring.

Let $C(X)$ be the number of alg-trees derivable by branch-paring from a given alg-tree rooted at x .

Each pruning of a descendant of x can be combined with the prunings of any other descendant of x to yield distinct alg-trees. We get

$$C(x) = C(x_1) * C(x_2) + 1$$

where x_1 and x_2 are the immediate descendants of x . The "1" is added to account for the alg-tree resulting when all nodes but x are pruned away. When x is a leaf, it has no descendants, so $C(\text{leaf}) = 1$.

This function suggests a rather large number of possibilities. In a binary, symmetric alg-tree, its value is greater than $\sqrt{2^n}$, where n is the number of non-leaf nodes in the tree. This motivated us to search for strategies which reduced the cost of searching this "tree-pruning" space.

The exponential nature of the dependence was based on the "independence" of the operation on each branch. Each branch must be pruned in all possible combinations with the prunings of other branches. We searched for a method which would decide how short any one branch should be, regardless of the remaining branches.

The fringe-sets a branch B gives rise to are ultimately set-united with fringe-sets arising from other branches. We hoped to avoid generating these "other branch" fringe-sets. We therefore searched for a criterion which would allow two interchangeable fv sets, A and B , both united with the same externally generated fv-set, C , to be compared. Specifically we need:

$$N(A \cup C) \geq N(B \cup C) \text{ for all } C.$$

Such a criterion was discovered. It promises to drastically reduce the number of fv-sets we need consider in each shape-set, by allowing

us to discard sets like A, which are known to be no better than a set, B, which we retain.

Tree Pruning Example:

We begin with:



a maximal alg-tree.

All its prunings are:



.

We compute $C(d)$, where d is the distance from the leaves of a node x in a binary symmetric tree.

$$C(d) = C(x).$$

We get

$$C(d) = [C(d-1)]^2 + 1$$

$$C(0) = 1$$

In contrast, the number of intermediate nodes in a binary symmetric tree of height d is $\phi(d)$, where

$$\phi(d) = 2\phi(d-1) + 1$$

$$\phi(0) = 0$$

Here, the height of a symmetric tree is the distance from its root to any of its leaves; intermediate nodes of a tree are non-leaf nodes. The number of operators in an expression equals the number of intermediate nodes of that expression's parse-tree.

$\phi(d)$	d	$C(d)$	
0	0	1	
1	1	2	
3	2	5	(Our example)
7	3	$26 = 5^2 + 1$	
15	4	$677 = 26^2 + 1$	

We will show that

$$C(d) > 2^{**[\phi(d-1) + 1]} \text{ for all } d \geq 2$$

and, since

$$\phi(d-1) + 1 = 1 + [\phi(d) - 1]/2 = [\phi(d) + 1]/2$$

that

$$C(d) > 2^{**[\phi(d)/2]} = (\sqrt{2})^{**\phi(d)}$$

Proof: by induction on d .

Case: $d = 2$.

$$C(2) = 5. \quad \phi(d-1) = \phi(1) = 1. \quad 2^{**}(\phi(d-1) + 1) = 2^{**}2 = 4$$

$$C(d) = C(2) = 5 > 4 = 2^{**}[\phi(1) + 1] = 2^{**}[\phi(d-1) + 1]$$

Case: $d > 2$. Assume the conclusion for $d - 1$.

$$C(d) = [C(d-1)]^2 + 1 > [C(d-1)]^2 > [2^{**}(\phi(d-2) + 1)]^2$$

$$C(d) > [2^{**}(\phi(d-2) + 1)]^2 = 2^{**}[2\phi(d-2) + 2]$$

$$2\phi(d-2) + 2 = [2\phi(d-2) + 1] + 1 = \phi(d-1) + 1$$

therefore

$$C(d) > 2^{**}[\phi(d-1) + 1]$$

holds for all $d \geq 2$.

Thus, if w = the number of operators in an expression, we will need to investigate somewhat more than

$$2^{**}[(w+1)/2]$$

tree prunings.

The comparison technique motivated by the "maximal alg-tree" algorithm can be profitably applied to the "leaves-in" algorithm. The effort required by the leaves-in algorithm is very similar to that required by the maximal alg-tree algorithm. It can be calculated as follows:

Let $D(x, S)$ = the number of alternative alg-trees rooted at x with root-shape S .

Then $D(\text{leaf}, S) = 1$, for each root-shape S .

At a non-leaf node, x , this number depends on the number of EEPT's with root-shape S , as well as the number of alg-trees in shape-set S_1 of descendant node x_1 .

When we choose EEPT K which matches E at x , a (shape, node) pair is determined for each leaf of K rooted at x . Let these pairs be

$$(S_{K1}, x_{K1}), \dots, (S_{KJ}, x_{KJ})$$

The number of combinations, each representing a possible alg-tree rooted at x choosable in this way is:

$$\prod_i D(X_{K1}, S_{K1})$$

Thus

$$D(X, S) = \sum_{K \in L} \prod_i D(X_{K1}, S_{K1}) + 1$$

where L indexes the EEPT's with root-shape at S matchable to E at X .

The term "+, 1" arises from the need to consider $n(x)$ a member of each shape-set, representing the computation of an array holding the result of x .

Let us assume that each EEPT K is a single-operator binary tree, so that its two leaves coincide with the sons of x when the EEPT is rooted at x .

Let us further assume that, for each shape S , there exists only one EEPT having a root-shape equal to S .

$$\text{Then } D(X, S) = D(X_1, S_{K1}) * D(X_2, S_{K2}) + 1$$

where X and X_2 are the sons of X .

Assuming that the tree is symmetric, and that

$$D(X, S_1) = D(X, S_j) \text{ for all shapes } S_1, S_j$$

$$\text{we have: } D(X, S) = D(X_1, S) * D(X_2, S) + 1,$$

$$D(\text{leaf}, S) = 1$$

or, in a symmetric tree containing W intermediate nodes

$$D(X, S) \geq 2^{**}(W/2)$$

CHAPTER IV

Introduction

Chapter III demonstrated that the cost of choosing an optimum compilation of a given matrix arithmetic expression appears to grow exponentially with the size of the expression. The present chapter is devoted to demonstrating a result which reduces this exponential dependence on the expression-size to linear dependence. The result, called the "comparison theorem", allows two interchangeable fv-sets (and hence, the alg-trees they represent) to be "compared".

Each alg-tree rooted at x was retained, in the leaves-in algorithm, to allow it to become a part of a "larger" alg-tree. The number of arrays this larger alg-tree requires depends not only on the alg-tree rooted at x , from which it was generated, but on the alg-trees rooted outside x which are also part of the larger alg-tree.

Suppose alg-trees A and B are both members of $G(S, x)$. Then whenever A can be parallel connected to some EEPT which matches E at y , some ancestor node of x , so can B . The number of arrays needed in computing y via A , together with some alg-trees C rooted at other descendants of y than x is $n(A \cup C)$. $N(A \cup C)$ is a major component of $n(A \cup C)$. The comparison theorem is capable of deciding, by examining only A and B , whether

$$N(A \cup C) \leq N(B \cup C)$$

without generating all the possible alg-trees C rooted outside x with which A and B might fuse. The comparison theorem itself gives necessary and sufficient conditions on A and B for the statement:

$$(1) \text{ for all } C \ N(A \cup C) \leq N(B \cup C)$$

to hold. These conditions are independent of C .⁵

⁵The trick of generalizing over a variable to derive conditions independent of that variable may work for other comparison predicates. This would suggest its use in exhaustive-searches. Sufficient "structure" must exist in the space being searched to allow a concept analogous to "interchangeable fringe-sets" to exist. Also, the comparison theorem in other searches may lack power (perhaps only holding between identical partial states) or applicability (perhaps few comparable pairs are ever produced). However, when proper conditions hold, it appears to be a powerful search-space reducing operator.

While the quantification makes this predicate independent of C , its determination would be too time consuming if every C had to be generated before the predicate could be computed. Thus, we seek a new predicate equivalent to (1), which does not specifically mention C .

In order to derive a predicate equivalent to (1), but not involving C , we investigate in some detail the evaluation rule for $N(S)$, where S is an fv-set. S is a set of integers (possibly including repetitions), S_1 . We have discovered that the function $N(S) = \max_{\substack{1 \leq i \\ S_i > 0}} (S_i + i - 1)$, where

$S_i \geq S_{i+1}$. $N(S)$ is then the maximum of a set of terms, $g(S,i)$, where $g(S,i) = S_i + i - 1$, $1 \leq i \wedge S_i > 0$. Not all these terms contribute directly to the maximum. Some, where $S[i_1] = S[i_2]$, merely act as placeholders, increasing the value of the index, i , but are themselves smaller than another term. In other words,

if $S_j = S_{j+1}$, then $g(S,j) = g(S,j+1) - 1 < g(S,j+1)$.

Hence, $\max_i(g(S,i)) > g(S,j)$.

Furthermore, the side condition requiring $S_i \geq S_{i+1}$ is not summarized in the term function, g , and must be handled separately.

We introduce a different method of computing $N(S)$, which eliminates both the need for the $S_i \geq S_{i+1}$ side condition, and emphasizes the "important terms" of the $g(S,i)$, i.e., those which may contribute to the maximum. They are characterized by $S_i > S_{i+1}$. The new method of computing $N(S)$ makes use of a new set of terms, $f(S,v) = I(S,v) + v - 1$. Rather than an index, v is a "value", an integer which may be found in the set S . $I(S,v)$ = the number of elements $S_i \geq v$. $I(S,v)$ incorporates the properties of the ordering condition $S_i \geq S_{i+1}$. It furthermore serves to give the index of the "important term" i whose $S_i = v$. Its extensions to values v not occurring in S introduces new "unimportant" terms (where $S_i > v > S_{i+1}$), but remains manageable. By making the definition of $f(S,v)$ conditional on $v > 0$ and $I(S,v) > 0$, so that $f(S,v) = 0$ where these conditions fail to hold, $N(S)$ can be computed by unrestrictedly maximizing $f(S,v)$ over v .

Once we have decided to use the sequences $f(S, v)$ for the computation of $N(S)$, we can discuss the effect on $N(S)$ of joining another set Y to S . Suppose Y is a set consisting of m copies of the integer x . Then

$$f(S \cup Y, v) = \begin{cases} f(S, v) & \text{if } v > x. \\ f(S, v) + m & \text{if } v \leq x. \end{cases}$$

In other words, joining a new set Y to S increases terms with low enough values by a constant amount. Except for cases when $x > S_1$ for all S_1 , no new important terms are introduced by the join. When Y is adjoined to two sets, S and T , we again have

$$\begin{aligned} f(S \cup Y, v) &= f(S, v) + m & \text{if } v \leq x \\ \text{and } f(T \cup Y, v) &= f(T, v) + m & \text{if } v \leq x \end{aligned}$$

Thus, if two sets S and T compare so that $N(S) \geq N(T)$, it may happen that for some sufficiently small v , v_0

$$f(S, v_0) < f(T, v_0).$$

Then, by adjoining some Y to both S and T , we can increase the values of the terms generated by v_0 in the new sets:

$$f(S \cup Y, v_0) = f(S, v_0) + m$$

$$f(T \cup Y, v_0) = f(T, v_0) + m$$

If the value u which minimizes $f(S, u)$ is larger than v_0 , then the terms generated by v_0 will, for large enough m , be larger than $f(S, u)$. When this happens, we will have

$$N(S \cup Y) < N(T \cup Y) \text{ for that } Y.$$

Example:

$$S = (41). \quad T = (212).$$

We must reorder S and T :

$$S' = (41), \quad T' = (221)$$

1	$g(S, 1)$	$g(T, 1)$
1	4	2
2	2	3
3	-	3

Thus, $N(S) = 4$, $N(T) = 3$, so $N(S) > N(T)$.

Suppose we adjoin $Y = (11)$ to both S and T and repeat the process of evaluation.

$$S \cup Y = (4111), T \cup Y = (21211)$$

$$(S \cup Y)' = (4111), (T \cup Y)' = (22111)$$

i	$g(S \cup Y, i)$	$g(T \cup Y, i)$
1	4	2
2	2	3
3	3	3
4	4	4
5	-	5

Now, $N(S \cup Y) = 4$, and $N(T \cup Y) = 5$. Thus Y has reversed the $S - T$ comparison.

In terms of the $f(S, v)$ and $f(T, v)$ representation, we have:

v	$f(S, v)$	$f(T, v)$	$f(S \cup Y, v)$	$f(T \cup Y, v)$
5	0	0	0	0
4	4	0	4	0
3	3	0	3	0
2	2	3	2	3
1	2	3	4	5

Notice that $f(S, 1) = 2$, while $f(T, 1) = 3$. So long as there is no $v \leq 1$ for which $f(S, v) > f(T, 1)$ (as there is not in this example) increasing $f(S, 1)$ and $f(T, 1)$ by a sufficiently large m will insure that $N(T) > N(S)$.

The following section derives more formally the predicate equivalent to (1) which does not refer explicitly to C . This predicate is abbreviated $A < B$ (or $B > A$). $B > A$ just when, for each integer $w > 0$, there is an integer v , satisfying $w \geq v > 0$, such that $f(B, v) \geq f(A, w)$.

Following the proofs of the equivalence of $B > A$ and (1), a section detailing the application of the comparison theorem to the leaves-in algorithm is presented. Here, we describe the "interchangeability" requirement conditions, as well as discussing the theorem's applicability

to comparisons of $n(B \cup C)$ to $n(A \cup C)$. It is shown that both fv-sets must be members of the same shape-set, as well as satisfying an inclusion condition on their fringe-shape-sets before one of the fv-sets can be discarded.

The remaining section of this chapter shows the power of the comparison theorem. Shape-sets produced during the leaves-in algorithm from EEPT's associated with the n^3 and $n^3/2$ basic algorithms have certain particularly useful properties. These properties allow advance prediction of the outcome of many fv-set comparisons between members of the same shape-set. Because the initial EEPT's satisfy an "equality" condition on their root and leaf-shapes, we can show that only two fringe-shape-set categories of fv-sets occur in each shape-set. Furthermore, we can show, using various properties of the function $f(S,v)$, that after $n(x)$ is added to each shape-set of node x , only one fv-set in each category will survive the comparisons. This serves to place a constant upper-bound on the number of fv-sets generated at each node, limiting the effort required by the search to a constant times the number of operators in the given expression.

IV.1 The Fv-Set Comparison Theorem

The letters A, B, and C here denote fv-sets, with $S(A)$, $S(B)$, and $S(C)$ their fringe sets.

Let $I(A, v)$ = the number of integers j in A such that $j \geq v$.

Properties of $I(A, v)$:

1. $w \geq v$ implies that $I(A, v) \geq I(A, w)$
since the set of values in A which are $\geq v$ contains the
set of values in A which are $\geq w$.
2. If $S(A)$ and $S(C)$ are disjoint, then $I(A \cup C, v) = I(A, v) + I(C, v)$.

Let $f(A, v) = I(A, v) + v - 1$ if $I(A, v) > 0$ and $v > 0$
= 0, otherwise.

Let $f(A) = \max_v f(A, v)$

Theorem 1: $f(A) = N(A)$

Let $A \leq B$ mean: $\forall w \exists v [w > 0 \rightarrow w \geq v > 0 \wedge f(B, v) \geq f(A, w)]$

Lemma 1: $A \leq B \rightarrow \forall C [A \cup C \leq B \cup C]$

Lemma 2: $A \leq B \rightarrow N(A) \leq N(B)$

Lemma 3: $\neg[A \leq B] \rightarrow \exists C [N(A \cup C) > N(B \cup C)]$

Theorem 2: $A \leq B \equiv \forall C [N(A \cup C) \leq N(B \cup C)]$

We follow this brief statement of the results of this section with their detailed proofs.

Suppose $A = (A[0], \dots, A[m], A[m+1]=0)$, with $A[i] \geq A[i+1] \geq 0$, for $m \geq i \geq 0$.

Properties of A:

1. If $v = A[i] > 0$, then $\exists j$ such that $v = A[j] > A[j+1]$.

Proof: There is at least one j such that $v = A[j]$, namely $j=i$.

Suppose $\forall j$ such that $v = A[j]$, $A[j] \leq A[j+1]$.

By construction of A, $A[j] = A[j+1]$.

Therefore $\forall j > i$ $v = A[j]$.

In particular $v = A[m+1] = 0$, contradicting
the assumption that $v > 0$.

2. If $A[j] \geq v > A[j+1]$ then $I(A,v) = j+1$.

Proof: $I(A,v)$ is the number of $A[j]$'s such that $A[j] \geq v$.

In constructing A , we get

$A[j] \geq$ exactly $j+1$ members of A ,

$A[0], \dots, A[j]$.

All of these are $\geq v$, so $I(A,v) \geq j+1$.

$v < A[j+1]$, so $v < A[i]$ for all $i \geq j+1$.

Hence, $I(A,v) = j+1$.

Theorem 1: $f(A) = N(A)$

where:

$$f(A) = \max_v f(A,v)$$

$$N(A) = \max_{\substack{m \geq i \geq 0 \\ A[i] > 0}} (A[i]+1, 0)$$

(where A is as before)

Let $g(A,i) = A[i]+1$ if $A[i] > 0$ and $m \geq i \geq 0$,
 $= 0$, otherwise.

Then $N(A) = \max_i g(A,i)$.

Proof:

1. If $A[i] > A[i+1]$ and $m \geq i \geq 0$, then $f(A, A[i]) = g(A,i)$.

Proof: $I(A, A[i]) = i+1 > 0$

if $A[i] = 0$, $f(A, A[i]) = 0 = g(A,i)$

else, $f(A, A[i]) = I(A, A[i]) + A[i] - 1$
 $= i + 1 + A[i] - 1$
 $= g(A,i)$

2. If $A[i] = A[i+1] > 0$ and $m \geq i \geq 0$ then $g(A,i) < g(A,i+1)$

Proof: $A[m+1] = 0$, so $m > i$, or $m \geq i+1$.

therefore, $g(A,i) = A[i]+1 = A[i+1]+1 < A[i+1]+i+1$

$g(A,i) < A[i+1] + i + 1 = g(A,i+1)$

3. If $g(A,i) = \max_j g(A,j) > 0$, then $A[i] > A[i+1]$ and $m \geq i \geq 0$

proof:

If $A[i] \rightarrow A[i+1]$, $A[i] = A[i+1]$.

Also, $g(A,i) > 0$, so $A[i] > 0$ and $m \geq i \geq 0$.

Then $g(A,i) < g(A,i+1)$, by 2.

But $g(A,i) = \max_j g(A,j) \geq g(A,i+1)$. Contradiction.

4. If $g(A,i) = \max_j g(A,j)$, then $g(A,i) = f(A,A[i])$.

proof:

By 1 and 3, if $g(A,i) > 0$. Otherwise $g(A,i) = 0$ for all i , $0 \leq i \leq m$, and $f(A,A[i]) = 0$, since $A[i] = 0$ for all i .

5. $\forall v \exists i f(A,v) \leq g(A,i)$.

proof:

If $A[i] \geq v > 0$ for some i such that $m \geq i \geq 0$, then $\exists j$ such that $A[j] \geq v > A[j+1]$.

therefore, $I(A,v) = j+1$, so

$$f(A,v) = I(A,v) + v - 1 = j + v \leq j + A[j] = g(A,j).$$

If $v \leq 0$ then $f(A,v) = 0 \leq g(A,0)$.

If $v > A[0]$ then $I(A,v) = 0$, so $f(A,v) = 0 \leq g(A,0)$.

We have: $\exists i$ such that $g(A,i) = \max_j g(A,j)$, and

$$f(A) \geq f(A,A[i]) = g(A,i) = \max_j g(A,j).$$

$$\forall v \exists i f(A,v) \leq g(A,i).$$

$$\text{For some } V, f(A) = f(A,V) \leq g(A,i) \leq \max_j g(A,j)$$

$$\text{Therefore } f(A) = \max_j g(A,j) = N(A).$$

Lemma 1A: If $v > 0$, then

if $I(A,v) > 0$, $f(A \cup C,v) = f(A,v) + I(C,v)$

and if $I(A,v) = 0$, $f(A \cup C,v) = f(C,v) \geq f(A,v) + I(C,v)$

proof:

Suppose $v > 0$.

Then $f(X,v) = I(X,v) + v - 1$ unless $I(X,v) = 0$.

We know that $I(A \cup C,v) = I(A,v) + I(C,v)$.

Then:

1. Suppose $I(A,v) > 0$. Then, since $I(C,v) \geq 0$,
 $I(A \cup C) > 0$.

$$\begin{aligned} \text{Therefore } f(A \cup C,v) &= I(A \cup C,v) + v - 1 \\ &= I(A,v) + I(C,v) + v - 1. \end{aligned}$$

$$\begin{aligned} \text{Also, } f(A,v) &= I(A,v) + v - 1, \\ \text{so } f(A \cup C,v) &= f(A,v) + I(C,v). \end{aligned}$$

2. Suppose $I(A,v) = 0$.

Then $I(A \cup C,v) = I(C,v)$.

Case: $I(C,v) = I(A \cup C,v) > 0$.

$$\begin{aligned} \text{Then } f(C,v) &= I(C,v) + v - 1 = I(A \cup C,v) + v - 1 \\ &= f(A \cup C,v) \end{aligned}$$

Also, $f(A,v) = 0$.

Therefore

$$\begin{aligned} f(A,v) + I(C,v) &= I(C,v) \leq I(C,v) + v - 1 \\ &\leq f(A \cup C,v). \end{aligned}$$

Case: $I(C,v) = I(A \cup C,v) = 0$.

Then $f(C,v) = 0 = f(A \cup C,v)$.

Also, $f(A,v) + I(C,v) = 0 \leq f(A \cup C,v)$.

Lemma 1B: If $v > 0$ then

$$f(A \cup C, v) \geq f(A, v) + I(C, v)$$

Proof: $I(A, v) > 0 \vee I(A, v) = 0$.

If $I(A, v) > 0$, $f(A \cup C, v) = f(A, v) + I(C, v)$ by Lemma 1A

$$\therefore f(A \cup C, v) \geq f(A, v) + I(C, v)$$

If $I(A, v) = 0$, $f(A \cup C, v) \geq f(A, v) + I(C, v)$ by Lemma 1A

$$\therefore f(A \cup C, v) \geq f(A, v) + I(C, v)$$

Lemma 1: $A < B \rightarrow \forall C [A \cup C < B \cup C]$

Recall that $A < B$ means $\forall w \exists v [v > 0 \rightarrow w \geq v > 0 \wedge f(B, v) \geq f(A, w)]$

Proof: We must show, assuming $A < B$, that for each $w > 0$ there

is a v' , $w \geq v' > 0$, such that

$$f(B \cup C, v') > f(A \cup C, w).$$

We know that for each $w > 0$ there is v such that

$$w \geq v > 0 \text{ and } f(B, v) \geq f(A, w).$$

Also, $w \geq v$ implies $I(C, v) \geq I(C, w)$,

$$\text{so } f(B, v) + I(C, v) \geq f(A, v) + I(C, w).$$

Case: $I(A, w) > 0$

Then $f(A, w) > 0$, so $f(B, v) > 0$, giving $I(B, v) > 0$.

$$\therefore f(A \cup C, w) = f(A, w) + I(C, w)$$

and $f(B \cup C, v) = f(B, v) + I(C, v)$, by Lemma 1A,

$$\text{so } f(B \cup C, v) \geq f(A \cup C, w),$$

and we may take $v' = v$.

Case: $I(A, w) = 0$.

Then $f(A \cup C, w) = f(C, w)$ by Lemma 1A

$f(C, w) \leq f(C, w) + I(B, w) \leq f(B \cup C, w)$, also by Lemma 1A.

Therefore, $f(A \cup C, w) \leq f(B \cup C, w)$

and we may take $v' = w$.

Lemma 2: $A < B \rightarrow N(A) \leq N(B)$

Proof: We show $A < B \rightarrow f(A) \leq f(B)$. (Then use Theorem 1.)

$$A < B \equiv \forall w \exists v [w > 0 \rightarrow w \geq v > 0 \wedge f(B, v) \geq f(A, w)]$$

$$\therefore \forall (w > 0) \exists v [f(B, v) \geq f(A, w)]$$

$$f(B) = \max_v f(B, v) \geq f(B, v)$$

$$\therefore \forall w > 0 \quad f(B) \geq f(A, w)$$

$$\text{also } f(A, 0) = 0, \text{ and } f(B) \geq 0$$

$$\therefore \forall w \quad f(B) \geq f(A, w)$$

$$\text{or } f(B) \geq f(A) = \max_w f(A, w) = f(A, w^*), \text{ for some } w^*$$

Lemma 3: $\neg[A < B] \rightarrow \exists c [N(A \cup C) > N(B \cup C)]$

$$\neg[A < B] \text{ means } \exists w \forall v [w > 0 \wedge [w \geq v > 0 \rightarrow f(B, v) < f(A, w)]]$$

Proof: If $N(A) > N(B)$, choose C empty.

Otherwise, for w as in (1) let

$$m = N(B) - f(A, w)$$

$$m \geq 0, \text{ for}$$

$$N(B) = f(B) \geq f(A) \geq f(A, w)$$

Take C to consist of $m + 1$ occurrences of the integer w .

$$\forall v \leq w \quad f(A \cup C, v) = f(A, v) + m + 1$$

$$\text{and } f(B \cup C, v) = f(B, v) + m + 1$$

$$\forall v > w \quad f(A \cup C, v) = f(A, v)$$

$$\text{and } f(B \cup C, v) = f(B, v)$$

$$\therefore f(A \cup C, w) = f(A, w) + m + 1 > N(B)$$

$$N(B) > f(B, v) = f(B \cup C, v) \text{ for all } v > w$$

$$\text{for all } v \leq w \quad f(A \cup C, w) = m + 1 + f(A, w) > m + 1 + f(B, v) = f(B \cup C, v)$$

$$\therefore f(A \cup C, w) > N(B \cup C)$$

$$\therefore N(A \cup C) > N(B \cup C)$$

Theorem 2: $A < B \equiv \forall c [N(A \cup C) \leq N(B \cup C)]$

Proof: $A < B \rightarrow \forall c [A \cup C < B \cup C]$

$$\rightarrow \forall c [N(A \cup C) \leq N(B \cup C)] \text{ by Lemma 2}$$

$$\neg[A < B] \rightarrow \neg \forall c [N(A \cup C) \leq N(B \cup C)] \text{ by Lemma 3}$$

$$\forall c [N(A \cup C) \leq N(B \cup C)] \rightarrow A < B$$

$$\therefore A < B \equiv \forall c [N(A \cup C) \leq N(B \cup C)]$$

Example: Use of the fv-set comparison theorem:

Given $A = (3221)$ and $B = (31111)$,

we investigate whether $A \succ B$, or $B \succ A$.

We will compute $f(A,v)$ and $k(A,v) = \max_{v \geq w > 0} f(A,w)$, as

well as $f(B,v)$ and $k(B,v)$.

We then need only ask if, for all $v > 0$, $f(A,v) \leq k(B,v)$ to determine if $A < B$.

v	$f(A,v)$	$k(B,v)$	$f(B,v)$	$k(A,v)$
4	0	5	0	4
3	3	5	3	4
2	4	5	2	4
1	4	5	5	4

For all v , $f(A,v) < k(B,v) \therefore A < B$.

We try $A' = (21)$, $B' = (3)$

v	$f(A',v)$	$k(B',v)$	$f(B',v)$	$k(A',v)$
4	0	0	0	2
3	0	3	3	2
2	2	2	2	2
1	2	1	1	2

Here, neither $A' \succ B'$, since $f(A',3) = 0 < k(B',3) = 3$

nor $B' \succ A'$, since $f(B',1) = 1 < k(A',1) = 2$

Potentially, if enough 'ones' are united with both A' and B' ,

$A' \cup C$ will eventually achieve a larger $N(A' \cup C)$ than will B' :

Let $C = (11)$. Then

$$A' \cup C = (2111) \quad , \quad N(A' \cup C) = \max(2+0, 1+1, 1+2, 1+3) = 4$$

$$B' \cup C = (311) \quad , \quad N(B' \cup C) = \max(3+0, 1+1, 1+2) = 3$$

Of course, $N(A') = 2$, $N(B') = 3$, so their "actual" situation can be reversed.

IV.2 Application of the Comparison Theorem to the Leaves-In Algorithm

Each fv-set produced during the leaves-in algorithm ultimately becomes part of an fv-set which is compared against all other fv-sets at some node. Fv-sets A which satisfy $n(A) > n(B)$ for some fv-sets A and B at node x are not chosen as the best method for computing x . If we had conditions which guaranteed that, for all C , $n(A \cup C) \geq n(B \cup C)$, and if each C joinable to A by some series of parallel connections were joinable to B as well, then A would not need further investigation. In particular, we would not have to generate fv-sets $A \cup C$ for each possible C , since we would know that an at least equally good alg-tree exists: $B \cup C$. Thus, A need not be retained in A 's shape-set at x so that generating all possible joins of A to C 's is avoided.

The previous section has shown that if (and only if) $A \succ B$, then for all C , $N(A \cup C) \geq N(B \cup C)$. We still do not know the relationship between $n(A \cup C)$ and $n(B \cup C)$, however. Furthermore, we must develop a criterion for the interchangeability of two fv-sets so that any C joinable to one can be joined to the other. The present section develops sufficient conditions for the application of the comparison theorem in deleting fv-sets from the shape-sets of the leaves-in algorithm.

First, we claim, by virtue of Property 1 of an alg-tree, that two fv-sets A and B which are both members of the same shape-set are interchangeable. For that alg-tree property shows that, if A can be a sub-alg-tree of some alg-tree, and is a member of $G(S, x)$ then so can any member B of $G(S, x)$. But $G(S, x)$ is just the shape-set containing A and B at x . Hence, A and B are interchangeable, if both belong to $G(S, x)$ for some node x and shape S .

The extension of the comparison theorem to $n(A \cup C)$ and $n(B \cup C)$ requires more thought. Basically, $n(A)$ may equal $N(A)$ or $N(A \cup 1)$, depending on whether A 's result can occupy an input array of A 's $AATA^6$, or not. This is determined by A 's fringe-shape set. The problem is that A may satisfy $N(A) \geq N(B)$, while $n(A) < n(B)$, for

⁶ Associated alg-tree algorithms. See Section 3, Chapter III.

example if $n(A) = N(A)$, while $n(B) = N(B \cup 1)$. The method used to join fringe-shape sets allows determination of conditions under which $n(A \cup C) \geq n(B \cup C)$ for all C .

Let C be an fv-set rooted at some node which is not x or a descendant of x . We say that C is an outside- x fv-set. If A is a member of some shape-set of x then when C is joined to A , the fringe-shape sets of A and of C set-unite. This fact allows us to derive conditions on the fringe-shape sets of A and of B , two fv-sets of the same shape-set, which guarantee that, if $A \succ B$, then $n(A \cup C) \geq n(B \cup C)$ for all C .

Let $F(C)$, where C is an fv-set be C 's associated fringe-shape set.

Theorem: If $A \succ B$, and $F(B) \supset F(A)$,
then $n(A \cup C) \geq n(B \cup C)$ for all C .

Proof: $F(A \cup C) = F(A) \cup F(C)$, by the steps of the leaves-in algorithm.
Therefore

$$F(B \cup C) = F(B) \cup F(C) \supset F(A) \cup F(C) = F(A \cup C)$$

Hence, if $A \cup C$ occurs in shape-set S , then if $S \in F(A \cup C)$, $S \in F(B \cup C)$. Therefore, $n(A \cup C) = N(A \cup C)$ implies that $n(B \cup C) = N(B \cup C)$.

Case: $n(A \cup C) = N(A \cup C)$. Then

$$\begin{aligned} n(B \cup C) &= N(B \cup C) \leq N(A \cup C) = n(A \cup C) \\ \text{so } n(B \cup C) &\leq n(A \cup C). \end{aligned}$$

Case: $n(A \cup C) = N(A \cup C \cup 1)$ Then

$$\begin{aligned} n(B \cup C) &\leq N(B \cup C \cup 1) \leq N(A \cup C \cup 1) \\ \text{so } n(B \cup C) &\leq n(A \cup C) \end{aligned}$$

In summary then, if A and B belong to the same shape-set, and $A \succ B$, and $F(B) \supset F(A)$, then A may be deleted from the shape-set without compromising the optimality of the compilation of the given expression into AATA's.

IV.3 The Leaves-In Algorithm, with Comparison Theorem

The following algorithm is to be applied to the nodes of the expression's parse-tree, E , in the following order. It is to be applied to a node x only after being applied to each of the descendant nodes of x , taken in any order.

- (1) If x is a leaf of E , then set $G(S, x) = [(0)]$ for each S . (0) is an fv-set constant, containing no shapes in its fringe-shape-set. Exit.
- (2) Initialize each shape-set $G(S, x)$ to the empty set.
- (3) Find an EEPT e which matches E at x . Suppose S is the root-shape of e . Find, for each leaf i of e , the node L_i of E corresponding to i in the match of e to E at x .
- (4) Select one member, for each i , of $G(\text{leaf-shape}(i), L_i)$. Join the selected combinations of fv-sets, using fv-set-join to combine the fv-sets, and set union to combine their fringe-shape sets. Add the resulting augmented fv-set to shape-set S .
- (5) Repeat (4) for each distinct combination of fv-sets selectable by (4).
- (6) Repeat (3)-(5) for each EEPT.
- (7) Calculate $n(x) = \min_i n(A_i)$, where A_i ranges over all fv-sets in any shape-set $G(S, x)$. Add the fv-set $(n(x))S$ to each shape-set $G(S, x)$. Here, $(n(x))S$ is an fv-set containing the integer $n(x)$ only, and whose fringe-shape set contains S . Replace $G(\emptyset, x)$ with $(n(x))$.
- (8) Compare each pair of fv-sets A and B in each shape-set $G(S, x)$. If

$$A \succ B, \text{ and } F(B) \supset F(A)$$
 then delete A from $G(S, x)$.
- (9) Exit.

The root of E will be the last node visited by this procedure. Along the way records can be kept, describing which fv-sets gave rise to each retained fv-set. The identity of the best alg-tree (fv-set) available for computing node x should be associated with some copy of $(n(x))$, say that which replaced $G(\Omega, x)$.

Apply the following algorithm to isolate each alg-tree whose AATA is part of the optimum compilation.

The following algorithm is applied first at the root of E.

- AATA(x): (1) Locate $G(\Omega, x)$. Collect, into set \underline{N} , all the nodes included in the alg-tree which $G(\Omega, x)$'s single member represents. This collection is accomplished by following the records of fv-set generation until the fringe-set nodes are reached. Let the fringe-set nodes be F_1 .
- (2) For each i , compute AATA(F_i).
- (3) Print \underline{N} , perhaps with additional information, indicating the EEPT rooted at each node in \underline{N} , and other information which is recorded in the fv-set tags. Exit.

IV.4 Leaves-In Algorithm Effort Requirement

In the following section, we demonstrate that, by restricting the given set of EEPT's appropriately, the effort required in applying the leaves-in algorithm to any given expression, E, is bounded by a linear function of the number of operators in E. We demonstrate this by showing that the comparison-and-deletion step of the leaves-in algorithm leaves no more than 2 fv-sets in each shape-set. Since no more than K shape-sets will appear at each operation node of E's parse-tree, no more than $2 \cdot K$ fv-sets occur at any node. Therefore, at each node, no more than $(2 \cdot K)^m$ fv-sets will be generated, where $m \geq$ number of leaves of any EEPT. This number is reduced to $2 \cdot K$ by less than $(2 \cdot K)^{(2 \cdot m)}$ comparison-and-deletion steps. Therefore, the operators of E, W in number,

generate approximately $W * [(2*K)^m + (2*K)^{(2*m)}]$ steps. Since K and m are constant with W , this shows that the effort is bounded by a linear function of W .

The critical step in our derivation of the linear effort-bound lies in bounding the number of fv-sets in any shape-set by 2. It is at this stage that we must impose a restriction on the given EEPT's.

Suppose we follow the steps of the leaves-in algorithm to a point just after all fv-sets have been computed for a given node. The next step involves computing

$$n(x) = \min_i n(a_i)$$

for all fv-sets a_i in any shape-set. The special fv-set $(n(x))$ is then added to each shape-set, representing the "null" alg-tree, a result stored in an intermediate array. We can show, under some circumstances, that $(n(x))$ and the comparison theorem reduce the number of fv-sets in each shape-set to 2, at most. If we consider only a finite number K of shapes, and hence shape-sets, this limits the number of fv-sets at each node to a constant $2*K$. Ultimately, this will let us show that if the expression contains W operators, only $(2*K) * W$ fv-sets are retained, at most. We thus bound the search effort.

We require (and will assume throughout this section) that all EEPT's satisfy:

Let $\text{root-shape}(e) = S$ and $\text{leaf-shape}(i,e) = T_i$,
for each leaf i of e .

For all leaves i of e ,

If $S \neq T_i$, then either $S = \Omega$ or $T_i = \Omega$.

The purpose of this restriction becomes clear in Theorem 1. Basically, the restriction guarantees that only one shape in each fringe-shape set can ever be relevant, regardless of the set-unions an fv-set enters. For each shape-set $G(S,x)$, that relevant shape in each of its member's fringe-shape sets is S . Furthermore, it ensures that the special fv-set $(n(x))$, which is added to each shape-set,

can be used to delete any fv-set in that shape-set. Here $n(x) = \min n(A)$ for all fv-sets A in shape-sets of node x . The restriction thus effectively relaxes the requirement that $F(B) \supset F(A)$ before B may be deleted by A . Theorem 1 partitions each shape-set into two classes. The remaining theorems show how $(n(x))$:

- (1) replaces one of these classes, and
- (2) leaves only elements B, C in the other such that $B \succ C$ and $C \succ B$.

Thus, we show that only one element remains in each class.

One final comment. The restriction we impose is light enough that all EEPT's generated by the n^3 algorithms we studied satisfy it. Furthermore, most of the $n^{3/2}$ algorithms also produce acceptable EEPT's. The theorems we prove here are thus not vacuous.

Theorem 1: If A is an fv-set in shape-set $G(S, x)$, then $F(A)$ as computed by the leaves-in algorithm satisfies:

If $S \neq \Omega$, and $T \neq S$ then $T \not\subseteq F(A)$.

Proof: By induction on $\text{level}(x)$. $\text{Level}(x)$ is an integer defined for each node x in the parse-tree E as:

$\text{level}(x) = 1 + \max \text{level}(x_i)$, for all sons x_i of x .

and $\text{level}(x) = 0$ if x has no sons (is a leaf of E).

When $\text{level}(x) = 0$, x is a leaf of E , and the fringe-shape-sets of all fv-sets A of all leaves are empty. Therefore, $T \not\subseteq F(A)$.

When $\text{level}(x) = I > 0$, we assume the the rem for all nodes y such that $\text{level}(y) < \text{level}(x)$. In particular, we assume it for all descendants x_i of x .

Each fringe-shape-set $F(A)$ in shape-set $G(S, x)$, $S \neq \Omega$, is generated by $F(A) = \bigcup_i F(A_i)$, where $F(A_i)$ is a fringe-shape-set in a shape-set $G(S_i, x_i)$ of some descendant node x_i of x . Furthermore, x_i is the j th leaf of the EEPT e rooted at x which generates $F(A)$. Also,

$$\text{root-shape}(e) = S$$

$$\text{leaf-shape}(j,e) = S, \text{ or (by the EEPT restriction,)} \\ = \Omega$$

If $\text{leaf-shape}(j,e) = S$, then $F(A_1)$ is chosen from a shape-set $G(S, X_1)$ such that $S \neq \Omega$.

But x_1 is a descendant of x , and by assumption, a fringe-shape-set in a shape-set $G(S, x_1)$ such that

$$S \neq \Omega \text{ satisfies}$$

$$\text{if } T \neq S \text{ then } T \notin F(A_1).$$

Also, if $\text{leaf-shape}(j,e) = \Omega$, $F(A_1)$ comes from shape-set $G(\Omega, x_1)$. But this shape-set's members all have empty fringe-shape-sets, so

$$T \notin F(A_1)$$

Therefore if $T \neq S$, then $T \notin F(A_1)$, so

$$T \notin \bigcup_1 F(A_1) = F(A)$$

A following step of the leaves-in algorithm replaces shape-set Ω at node x with the fv-set $(n(x)) = X$. When added to shape-set Ω , $F(X) = \emptyset$. Also, X is adjoined to shape-set $G(S, x)$, with $F(X) = S$. After this step, if $T \neq S \neq \Omega$, and if B is an fv-set in shape-set S , then $T \notin F(B)$.

Thus, the theorem is true for nodes x such that

$$\text{level}(x) = 1, \text{ and hence true for all nodes } x \text{ in } E.$$

Corollary: If A is an fv-set in shape-set $G(S, x)$, and $S \neq \Omega$, then $F(A)$ is either empty, or contains only S .

Proof: If $F(A)$ contained $T \neq S$, it would violate Theorem 1 of this section.

Thus, we can divide the fv-sets A in shape-set $G(S, x)$ into two disjoint classes, those such that

$$F(A) = \{S\}$$

and those such that

$$F(A) = \emptyset$$

The first class will be called "1-class", the second "0-class".

During the leaves-in algorithm, for each node x we compute $n(x)$, and adjoin $(n(x))$ to each shape-set of x . $X = (n(x))$ becomes a part of 1-class of each non- Ω shape-set. We will show that all members B of a given 1-class satisfy $B \succ X$. Since B and X are both members of the same shape-set, $G(S, x)$, and $F(B) = F(X) = S$, B is deletable by X . Therefore, after the deletion, only one member, X , is left in each 1-class. Similarly, we can show that only one member is left in each 0-class. This demonstrates that in each shape-set, only 2 members remain after the comparison-and-deletion step of the leaves-in algorithm.

Theorem 2: If n is an integer ≥ 0 , and B an fv-set, then if
 $n \leq N(B)$, $(n) \prec B$.

Proof: $N(B) \geq n$ implies that

$$\forall v \ f(B, v) \geq n.$$

Also, (n) has the property that

$$f((n), w) \leq n \text{ for all values } w,$$

since if $w > n$, $I((n), w) = 0$

$$\text{so } f((n), w) = 0 \leq n$$

and if $n \geq w > 0$, $I((n), w) = 1$

$$\text{so } f((n), w) = I((n), w) - 1 + w = w \leq n.$$

Of course, if $w = 0$, $f((n), w) = 0 \leq n$.

Therefore, $\forall v, w \ f(B, v) \geq n \geq f((n), w)$.

In particular, for each $w > 0 \ \exists v$ such that

$$w \geq v > 0 \text{ and}$$

$$f(B, v) \geq f((n), w)$$

therefore, $B \succ (n)$.

Theorem 3: $(n(x)) \leq A$ for all fv-sets A in any 1-class.

Proof: By definition $n(x) = \min_i (n(A_i))$ for all fv-sets A_i at node x .

Therefore $n(x) \leq n(A) = N(A)$ for all A in any 1-class.

Therefore $(n(x)) \leq A$, by theorem 2.

Theorem 4: If B is an fv-set in 0-class, and $B > (n(x))$, then B will be deleted.

Proof: $F(B)$ is empty, by definition of 0-class. The 1-class of the shape-set containing B contains $X = (n(x))$, and hence both $(n(x))$ and B belong to the same shape-set. Also, $F(X) \supset F(B) = \emptyset$. Therefore, B is deletable by X . Hence, if $B > (n(x))$, B will be deleted by the leaves-in algorithm.

Suppose B is an fv-set of 0-class which remains after the deletion step.

Lemma 1: $N(B) < n(x)$.

Proof: If $N(B) \geq n(x)$, then $B > (n(x))$ by Theorem 2, and would be deleted, by Theorem 4.

Lemma 2: $N(B \cup 1) > N(B)$.

Proof: $n(B) = N(B \cup 1)$, since $B \in 0$ -class

$$n(B \cup 1) = n(B) \geq n(x) > N(B).$$

Theorem 5: If $N(C \cup 1) > N(C)$ then $f(C, 1) = N(C)$,
and $N(C \cup 1) = N(C) + 1$.

Proof: $N(C) = \max_v f(C, v)$

$$N(C \cup 1) = \max_v f(C \cup 1, v)$$

$$\begin{aligned} f(C \cup 1, v) &= f(C, v) + I(1, v) \\ &= f(C, v) \text{ if } v > 1, \\ &= f(C, 1) + 1 \text{ else.} \end{aligned}$$

$$\therefore \forall v > 1 \quad f(C \cup 1, v) = f(C, v), \text{ while}$$

$$f(C \cup 1, 1) = f(C, 1) + 1$$

If $f(C,1) < N(C)$, then:

$$1 + f(C,1) \leq N(C).$$

$$\therefore \forall v \ f(C \cup 1, v) \leq N(C),$$

or $N(C \cup 1) \leq N(C)$ contradicting the theorem's hypothesis.

Therefore, $f(C,1) \geq N(C)$.

$$\text{But } f(C,1) \leq \max_v f(C,v) = N(C)$$

$$\text{so } f(C,1) = N(C).$$

$$\text{Also, for all } v, \ f(C,v) \geq f(C \cup 1, v) - 1$$

$$\text{In particular } f(C, v^*) \geq f(C \cup 1, v^*) - 1 = N(C \cup 1) - 1,$$

$$\text{so } N(C) + 1 \geq N(C \cup 1)$$

$$\text{also, } N(C \cup 1) > N(C), \text{ so } N(C \cup 1) \geq N(C) + 1$$

$$\text{giving } N(C \cup 1) = N(C) + 1.$$

Theorem 6: If $N(C) = f(C,1) = f(B,1) = N(B)$,

then $C \succ B$ and $B \succ C$.

Proof: $f(C,1) = N(B) = \max_v f(B,v)$

Therefore, $f(C,1) \geq f(B,v)$, for all v

Thus, for all $v > 0$, $w = 1$ satisfies

$$v \geq w > 0 \text{ and}$$

$$f(C,w) = f(C,1) \geq f(B,v).$$

Therefore $C \succ B$.

Similarly, because $f(B,1) = N(C)$, $B \succ C$.

Theorem 7: If C remains in 0-class after the deletion step,

$$N(C) = n(x) - 1, \text{ and}$$

$$N(C) = f(C,1).$$

Proof: By Lemma 2 $N(C \cup 1) > N(C)$, so

by Theorem 5 $N(C \cup 1) = N(C) + 1$.

Also, by Lemma 1, $N(C) < n(x)$,

so $n(x) \leq N(C \cup 1) = N(C) + 1$.

We have $N(C) < n(x) \leq N(C) + 1$ for integers

$N(C)$, $n(x)$, so

$N(C) = n(x) - 1$.

Part 2 follows directly from Lemmas 1,2 and Theorem 5.

Theorem 8: If B and C remain in 0-class after the deletion step,
 $B \succ C$ and $C \succ B$. Therefore one may be deleted.

Proof: By Theorem 7, $N(B) = n(x) - 1 = N(C)$.

Also by Theorem 7, $N(B) = f(B,1)$ and $N(C) = f(C,1)$.

Therefore by Theorem 6, $B \succ C$ and $C \succ B$.

Since both B and C are in 0-class, they are in the same
 shape-set, and $F(B) = F(C) = \emptyset$.

Therefore, one may be deleted.

An immediate consequence of Theorem 8 is that only one fv-set
 remains in 0-class after the deletion step. Also, Theorem 3 has shown
 that only one fv-set remains in 1-class. Since 0-class and 1-class of
 a shape-set together cover that shape-set, only 2 fv-sets remain in
 each shape-set after the deletion step.

CHAPTER V

V.1 Summary of Results

We have described a transformation, loop-fusion, on programs. If X is a sequence of two loops satisfying certain conditions on the sets of variables accessed, $\text{loop-fusion}(X)$ is a single loop, computationally equivalent to X , which executes no more operations than X . Such equivalent, time-conservative transformations, applied to any program, yield new programs, with different characteristic space requirements, which are valid alternative programs for the programming task performed by the given program. We can search the space of such programs for one which requires least space.

We present two sets of alternative programs for computing the matrix assignment statements $C \leftarrow A * B$ and $C \leftarrow A + B$. Each set of programs forms the basis for an "equal-time" collection of algorithms for evaluating matrix arithmetic expressions in $+$ (matrix addition) and $*$ (matrix multiplication) on square N -by- N matrices. Each compilation of a given matrix arithmetic expression into sequences of algorithms in a given equal-time collection of algorithms requires the same amount of execution time. These equal-time collections are derived by loop fusion from the set of algorithms forming the basis for the collection.

An algorithm for choosing that compilation of any given matrix arithmetic expression, E , which uses the fewest 2-arrays⁷ is presented. This algorithm, called the leaves-in algorithm, uses the properties of loop-fusion to "tailor" algorithms, selected from an equal-time collection C , to fit each part of E . It searches over all possible compilations of E into algorithms of C , potentially generating a number of algorithms which is proportional to $\sqrt{2}^{**}W$, where there are W operators in E .

A general technique is then presented for reducing the number of cases which an exhaustive search for an optimum alternative must examine.

⁷ A 2-array is a set of variables capable of holding one N -by- N matrix.

Such searches can generate alternatives by assigning values one by one to the state-variables which describe an alternative. Usually, the value of the criterion function of a partially-specified alternative cannot be computed without completing the specification in all possible ways. The given technique allows some partially-specified alternatives to be rejected without generating all completions, by guaranteeing that for every completion $C \cup A$ of one such alternative, A , there is a completion $C \cup B$ of another, B , which is better than $C \cup A$. Thus, generating all the completions $C \cup A$ of A is unnecessary in the search for an optimum-valued alternative.

This technique is applied to the search performed by the leaves-in algorithm. Here, an "alternative" is an algorithm for evaluating the entire expression, E . The value of the criterion function, $N(S)$, of an alternative S is the number of 2-arrays needed by S . A partially-specified alternative A is an algorithm for evaluating some subexpression $E(A)$ of E . A predicate $P(A,B)$ is defined on partially-specified alternatives A and B , equivalent to

$$\forall C \quad N(A \cup C) \geq N(B \cup C),$$

where $A \cup C$ is a completion of A , and $B \cup C$ is a completion of B , derived by using B instead of A in subexpression $E(A)$. $P(A,B)$ may be evaluated without generating all alternatives C . When $P(A,B)$ is true, A may be rejected without investigating all its possible completions, for each is known to be no better (i.e., no smaller) than some completion of B .

Using predicate $P(A,B)$ to reject alternatives generated during the leaves-in algorithm, a modified leaves-in algorithm is produced. This modified algorithm investigates only $k * W$ alternatives, where W is the number of operators in the expression E , and where k does not depend on W .

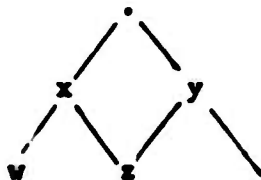
We have studied a set of program alternatives for implementing one class of matrix arithmetic expressions, searching for a program which uses fewest 2-arrays while never computing the value of any element of a subexpression more than once. We must admit that not all programs satisfying these criteria have been investigated. In particular, we have

studied only those programs derivable by loop fusion. Other methods of constructing algorithms for evaluating matrix arithmetic expressions may exist, possibly yielding algorithms which use fewer 2-arrays than those the leaves-in algorithm can discover. Nevertheless, searches over "program technologies" like that which the leaves-in algorithm investigates are interesting in their own right, and may well yield near-optimum results.

Two types of generalization of our work come to mind. Certainly, different optimization criteria could be used, in particular, allowing program combinations which are not minimum-connection-time, and optimizing some combination of program execution time and memory space. Also, many generalizations of "matrix arithmetic expressions" as we have defined them appear interesting. We feel that it may be worthwhile to indicate some of the possible expression generalizations which our current technique cannot handle.

First, we could consider expressions containing more than one occurrence of a particular subexpression. The data flow diagram of such an expression contains nodes having more than one incident line. The presence of such nodes makes $n(x)$, for some nodes x in the diagram, dependent on more than $n(x_1)$ for all descendants x_1 of x .

For example, consider:



the value of $n(x)$ depends on whether y is to be computed before, or after x . If before, then the common node, z , must be computed before w . If after, the orders $[w; z]$ and $[z; w]$ are both possible.

Secondly, we could allow use of the associative laws of matrix addition and multiplication by relaxing the requirement that an expression be fully parenthesized. Here, each possible association could be

generated, and the leaves-in algorithm could be applied to each resulting binary parse-tree. A more elegant, less time-consuming search should be devised, however.

Thirdly, we could consider allowing the variables of the expression to be rectangular matrices with the usual conformability requirements imposed. This generalization we believe lies in the scope of the leaves-in algorithm. The major requirement is a generalization of the definition of fv-set to allow arrays of various sizes to hold the different inputs to an associated alg-tree algorithm.

Other programming language constructs far different from matrix arithmetic expressions could conceivably be technologically optimized. These constructs must be such that several alternative implementations are available for each instance of the construct.

One of the more interesting of such examples, in which the available alternatives are particularly clear, concerns constructs which specify parallelism. These constructs indicate to a compiler that certain operations can "proceed in parallel", i.e., that any ordering of these operations which preserves the relative order of the operations in each "parallel sequence" yields equivalent results. These constructs are intended for an environment in which more than one processor is available. However, where only one processor exists, they permit the compiler to choose a space-minimal ordering of the given operations.

An important class of problems, with consequences for program optimization, concerns the development of transformations which generate programs computationally equivalent to a given program. Such transformations may involve change of data representation, change of sequence of certain operations, or more drastic changes, making use of mathematical properties of the programming task description of the task the program implements. We have presented one such transformation (loop fusion). We have made use of another, re-ordering of operation sequences, in generating the basic n^3 algorithms for matrix addition and multiplication. Other transformations exist. Notably, we can consider developing techniques for compiling functions, described as collections of recursive subroutines, into efficient

iterative programs.

At another level, there may well exist program communication alternatives whose value can best be investigated by an exhaustive search. For example, variables in a program may exist which the programmer has allocated separately, for conceptual reasons, but whose contents are never relevant simultaneously. Compilers could locate and combine these.

One could also conceive of automatic choices being made of alternative numerical procedures for various phases of certain programs. A search procedure, in conjunction with an automatic error analysis, may be useful, to determine the actual sensitivity of the results to small perturbations in the input values. Such an "experimental" approach, combined with alternative numerical procedure trials, might yield smaller error bounds than can conventional human-implemented numerical analysis.

The general problem of program optimization is difficult for several reasons. First, the number of possible programming approaches to many interesting programming tasks seems to be extremely large. The size of this number prohibits an exhaustive generation of each possible program capable of performing the given task. Second, the possible programming alternatives for a given task are difficult to determine. This is partly the fault of the languages in which these tasks are programmed. These languages often require that the programmer specify more details of the procedure to be followed than are essential to the task to be performed. Third, a given programming task must be optimally programmed not once, but many times. Each time that task appears as a subtask of some larger programming task, the program which implements it optimally must change to best fit the new context. Thus, one cannot hope to produce an optimized program for all task contexts. One could profitably develop algorithms for rapidly finding such optimum programs. The hope of producing such programmer-aided algorithms motivated this study.

The work reported here has barely brushed the surface of the study of efficient programs. We have gained some insight into only a few of the devices programmers use so freely in producing their programs. The

possibility of reducing programmer effort by providing programming algorithms, rather than rules-of-thumb, motivated our study. We feel that many additional interesting and useful programmer-aiding algorithms remain to be discovered.

APPENDIX I

1. Winograd's Matrix Multiply Algorithm [14]

The following algorithm, due to S. Winograd, can also be used to compute the matrix assignment statement $C = A * B$. We present a derivation of the algorithm, and the counts of the number of scalar additions and multiplications it requires.

$$C_{kj} = \sum_{i=1}^n x_i y_i, \text{ where } x_i = A_{ki} \text{ and } y_i = B_{ij}.$$

yields $C = A * B$, where A , B and C are (square) matrices.

n even:

$$\sum_{i=1}^n x_i y_i = \sum_{i=1}^{n/2} (x_{2i-1} y_{2i-1} + x_{2i} y_{2i}).$$

$$(x_{2i} + y_{2i-1})(x_{2i-1} + y_{2i}) = x_{2i-1} y_{2i-1} + x_{2i} y_{2i} + x_{2i} x_{2i-1} + y_{2i} y_{2i-1}$$

$$\text{Therefore, } \sum_{i=1}^n x_i y_i = \sum_{i=1}^{n/2} (x_{2i} + y_{2i-1})(x_{2i-1} + y_{2i})$$

$$= \sum_{i=1}^{n/2} x_{2i} x_{2i-1} + \sum_{i=1}^{n/2} y_{2i} y_{2i-1}.$$

Of the terms on the right, the last two need be computed only once for each row of A or column of B . Thus, the operation counts are:

$$*: n^2 * \frac{n}{2} + n(\frac{n}{2} + \frac{n}{2})$$

$$+: n^2 (\frac{3n}{2}) + n(\frac{n}{2} + \frac{n}{2})$$

$$\text{or } *: n^3/2 + n^2$$

$$+: 3n^3/2 + n^2$$

n odd:

$$\sum_{i=1}^n x_i y_i = x_n y_n + \sum_{i=1}^{(n-1)/2} (x_{2i} + y_{2i-1})(x_{2i-1} + x_{2i}) \\ - \sum_{i=1}^{(n-1)/2} x_{2i} x_{2i-1} - \sum_{i=1}^{(n-1)/2} y_{2i} y_{2i-1}.$$

The operation counts in this case are:

$$*: n^2 + n^2 \cdot (n-1)/2 + n \left[\frac{(n-1)}{2} + \frac{(n-1)}{2} \right]$$

$$+: n^2 + n^2 \cdot \frac{3(n-1)}{2} + n[n-1]$$

$$\text{or } *: n^3/2 + 3n^2/2 - n$$

$$+: 3n^3/2 + n^2/2 - n$$

2. Additional Shapes Defined for Winograd's Algorithm:

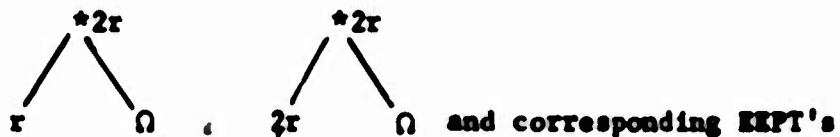
2r - double row - All $[2I, x]$ and $[2I-1, x]$

for $1 \leq x \leq N$

2c - double column - All $[x, 2I]$ and $[x, 2I-1]$ satisfying $1 \leq x \leq N$

3. Variations on Winograd's Algorithm (N Even)

The variations on the basic algorithm presented here by no means exhaust the useful versions of the $n^3/2$ matrix multiplication algorithm. The introductions of the shapes 2r and 2c suggests a still larger class of algorithms with KEPT's like:



for 2c and c.

(1) $I \rightarrow N$

$U[I] \leftarrow 0$

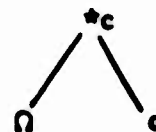
$A: n$

$K \xrightarrow{2} N$

$B: c$

$U[I] \leftarrow A[I, K] * A[I, K-1]$

$C: c$



$J \rightarrow N$

$V \leftarrow 0$

$K \xrightarrow{2} N$

$V \leftarrow B[K, J] * B[K-1, J]$

$I \rightarrow N$

$C[I, J] \leftarrow -U[I] - V$

$K \xrightarrow{2} N$

$C[I, J] \leftarrow (A[I, K] + B[K-1, J]) * (A[I, K-1] + B[K, J])$

(2) $I \rightarrow N$ $V[I] \leftarrow 0$ $A: 2c$ $U[I] \leftarrow 0$ $B: 2r$ $J \rightarrow N$ $C: 0$  $C[I, J] \leftarrow 0$ $K \xrightarrow{2} N$ $I \rightarrow N$ $U[I] + \leftarrow A[I, K] * A[I, K-1]$ $V[I] + \leftarrow B[K, I] * B[K-1, I]$ $J \rightarrow N$ $C[I, J] + \leftarrow (A[I, K] + B[K-1, J]) * (A[I, K-1] + B[K, J])$ $I \rightarrow N$ $J \rightarrow N$ $C[I, J] - \leftarrow U[I] + V[J]$

4. Combining two copies of (1) to compute $E \leftarrow D * (A * B)$:

$I \rightarrow N$

$U[I] \leftarrow 0$

$K \xrightarrow{2} N$

$U[I] + \leftarrow A[I,K] + A[I,K-1]$

$J \rightarrow N$

$V \leftarrow 0$

$A: \Omega$

$K \xrightarrow{2} N$

$B: c$

$V + \leftarrow B[K,J] * B[K-1,J]$

$\underline{C}: c$

$I \rightarrow N$

$C[I,J] \leftarrow -u[I] - V$

$K \xrightarrow{2} N$

$C[I,J] + \leftarrow (A[I,K] + B[K-1,J]) * (A[I,K-1] + B[K,J])$

$I \rightarrow N$

$W[I] \leftarrow 0$

$D: \Omega$

$K \xrightarrow{2} N$

$C: c$

$W[I] + \leftarrow D[I,K] * D[I,K-1]$

$\underline{E}: c$

$J \rightarrow N$

$X \leftarrow 0$

$K \xrightarrow{2} N$

$X + \leftarrow C[K,J] * C[K-1,J]$

$I \rightarrow N$

$E[I,J] \leftarrow - W[I] - X$

$K \xrightarrow{2} N$

$E[I,J] + \leftarrow (D[J,K] + C[K-1,J]) * (D[I,K-1] + C[K,J])$

APPENDIX II

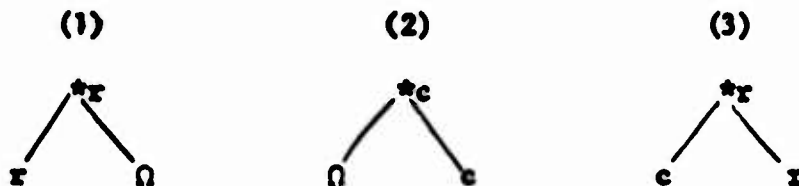
Leaves-In Algorithm in APL

We present here a program, written in APL,⁵ which demonstrates the "leaves-in" algorithm. This algorithm describes how a given expression's parse-tree can best be computed by alg-tree associated algorithms (AATA's). Each AATA is grown by parallel connection from a set of KEPT's, representing a set of elementary algorithms, which the user supplies. The "best" method of computing the given expression is that composition of AATA's which uses the fewest arrays for communicating results from one AATA to inputs of another.

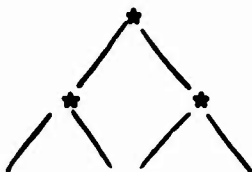
The inputs to the program describe the KEPT's tree structure, and the shapes and operators associated with their nodes. Also, the structure of the expression's parse-tree is given. The result is a list of alg-trees, whose AATA's are a series of elementary algorithms which, executed in the given order, produce the required expression value. Each alg-tree is described by listing the nodes in the tree which it includes, the KEPT rooted at each node, and the intermediate array assigned to hold each input, and the result. All KEPT's listed in one alg-tree are to be fused. The root of the alg-tree is always listed first, and is associated with the number of the intermediate array which is to hold the AATA's result.

As an example, we describe the KEPT's of the N^3 algorithms, and their optimal assignment to the parse-tree of the expression $(A*B) * (C*D)$.

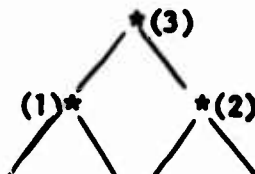
KEPT's:



Parse-Tree:



Result:
a single alg-tree:



Uses one temporary to hold the final result.

⁵APL is a conversational language, developed by K. E. Iverson, L. M. Breed, and R. H. Lathwell for the IBM 360/50. The language is described in [2].

The internal structure of the APL program is notable for its use of the effort-limiting results we presented. The "core" of the method lies in the comparison of generated fv-sets. We retain a "current value" of $n(x)$ throughout our generation of fv-sets at node x . Since we know that $n(x)$ is the only element of the 1-set of each shape-set, we do not copy it. Furthermore, space for only one member of each non- Ω shape-set (the single retained 0-set member) is reserved. As an fv-set is created, it is tested to see if it reduces the current value of $n(x)$, and, if it is a 0-set member, to see if it will be retained in the 0-set. These comparisons never require the actual fv-set comparison algorithm. Because of the way we represent fv-sets, and retain values $n(A)$, for certain fv-sets A , the comparisons are between single integers only.

External Representations:

(1) Inputs:

(a) Tree structure:

The structure of a tree is input in a single vector called 'FATHERS'. $FATHERS[i] = j$, where node i has father j in the tree. In labeling the nodes of a tree, the father of i must be given a number greater than i , so that FATHERS must satisfy $FATHERS[i] > i$. Furthermore, left siblings must be numbered less than their right siblings. (If these rules are violated, the results are unpredictable.)

The FATHERS-entry for the root of the tree is not part of the tree-structure. It must be present, but its value carries non-structural information.

(b) Node labels:

The internal label of a node i , is given by a code number, K_i , in the i th position of vector OPERATORS. Codes are used to indicate the operator ($*$ or $+$) for intermediate nodes, or, in the case of leaves of EEPT's, to indicate shape (Ω , r , or c). Since the root-node of

an EEPT has an operator, the EEPT's root-shape is coded in place of the FATHER of the EEPT's root.

Codes:

0 - variable
1 - *
2 - +

Shapes:

1 - Ω
2 - π
3 - c

(2) To input EEPT's type:

EEPTS

The program responds with alternate requests for FATHERS, and OPERATORS, which should be answered with the appropriate vectors. Each pair of requests allows the input of another EEPT. EEPT's are identified by the order of their input, the first one being given an identification number of '1'. Any scalar or single-element vector typed in response to FATHERS is ignored, and terminates the input of EEPT's.

To execute the leaves-in algorithm, after EEPT's have been input, type:

TREE

The response is a FATHERS, OPERATORS request-pair which should be answered with the parse-tree description. The leaves-in algorithm then executes.

(3) Output:

The output is a sequence of alg-trees, assignments of EEPT's to the nodes of the parse-tree. Each alg-tree is given in three vectors:

NODES[I] - lists the node-number of the node in the parse-tree

associated with the root of each KEPT, and with each KEPT leaf. NODES[I] always lists the root of the alg-tree.

KEPTS[I] - the identifying number of the KEPT associated with node NODES[I] in this alg-tree.

TEMPS[I] - the number of a temporary matrix assigned to hold a result associated with node NODES[I], or zero.

The sequence in which alg-trees are listed is the sequence in which the algorithms they represent are to execute. This ensures that an intermediate matrix used as input to a given AATA is computed before it is accessed.

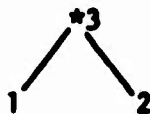
Detailed examples:

A. Input of KEPT's:

1. F: 3 3 2
O: 2 1 1



Here the tree is labeled



Its structure, given in F: (fathers), is

3 3 _

signifying that nodes 1 and 2 have father '3', and reserving the last position of the vector (which always represents the "father" of the tree's root) for other information.

The operators are __ 1, i.e., node 3 has operator 1 = *, and the other nodes, known to be leaves of the tree, have no operators.

The additional information given in the two vectors represents the shape associated with each node:

Shapes:

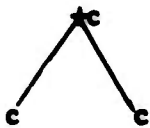
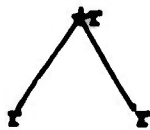
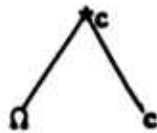
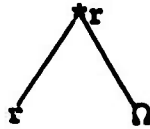
F: _ _ 2

O: 2 1 _

Thus, nodes 3 and 1 have shape 2 = r , while node 2 has shape 1 = Ω .

- B. Input example: Entry of KEPT's.
The indented line following the □: line is typed by the user, not the computer.

EEPTS	
FATHERS	
□:	
3 3 2	
OPERATORS	
□:	
2 1 1	
FATHERS	
□:	
3 3 3	
OPERATORS	
□:	
1 3 1	
FATHERS	
□:	
3 3 1	
OPERATORS	
□:	
3 2 1	
FATHERS	
□:	
3 3 2	
OPERATORS	
□:	
2 2 2	
FATHERS	
□:	
3 3 3	
OPERATORS	
□:	
3 3 2	
FATHERS	
□:	
0	



C. Output interpretationExample 1:

```

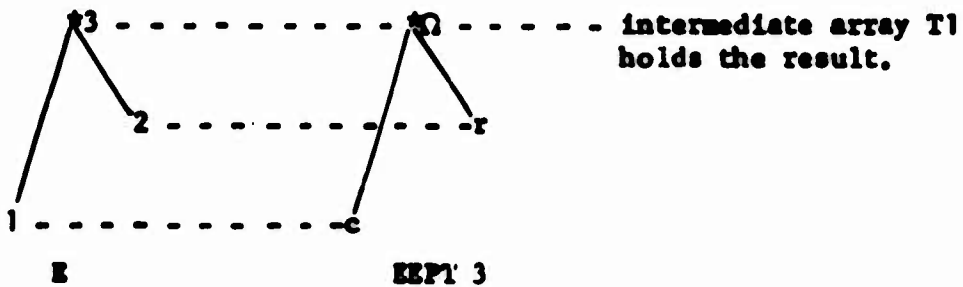
      TREE
FATHERS
□:
    3 3 0
OPERATORS
□:
    0 0 1
ASSIGNED 1 MATRICES:
NODES: 3 2 1
EEPTS: 3 0 0
TEMPS: 1 0 0

```

the input, FATHERS and OPERATORS, describes the tree E:

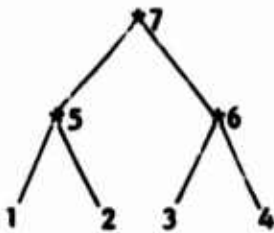
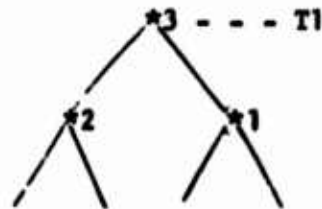
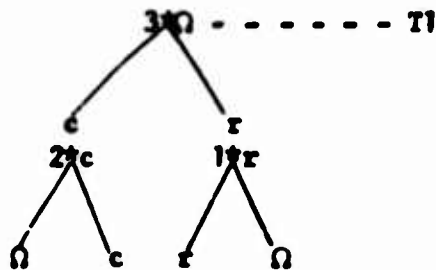


The output (starting with the line reading "ASSIGNED 1 MATRICES") gives the number of MATRICES (2-arrays) needed in computing the root of E. The remainder of the output presents the alg-tree(s) whose AATA(s) correctly compute E. In this case, a single alg-tree suffices:



Example 2:

TREE
 FATHERS
 []:
 5 5 6 6 7 7 0
 OPERATORS
 []:
 0 0 0 0 1 1 1
 ASSIGNED 1 MATRICES:
 NODES: 7 6 4 3 5 2 1
 EEPTS: 3 1 0 0 2 0 0
 TEMPS: 1 0 0 0 0 0 0

tree E:**one alg-tree:****EEPT's used:**

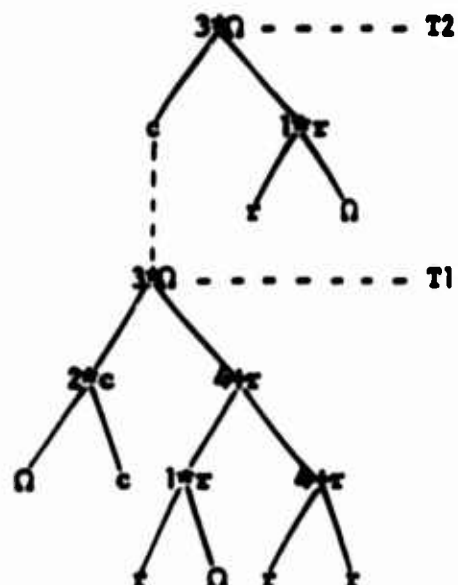
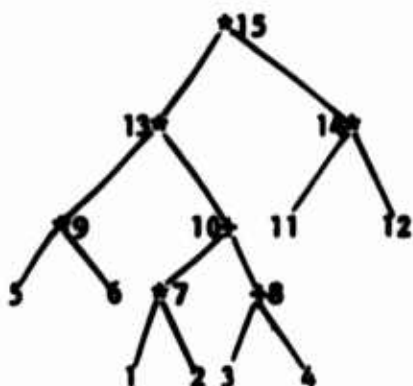
Example 3:

```

TREE
FATHERS
□:
  7 7 8 8 9 9 10 10 13 13 14 14 15 15 0
OPERATORS
□:
  0 0 0 0 0 0 1 2 1 2 0 0 1 1 1
ASSIGNED 2 MATRICES:
NODES: 13 10 8 4 3 7 2 1 3 6 5
EEPTS: 3 4 4 0 0 1 0 0 2 0 0
TEMPS: 1 0 0 0 0 0 0 0 0 0 0
NODES: 15 14 12 11 13
EEPTS: 3 1 0 0 0
TEMPS: 2 0 0 0 1

```

tree:



E. Internal Representations of Interest:**(1) Fringe-value set.**

The contents of each fringe-value set S is represented as a table of $f(S,v)$, $v = 1, 2, 3, 4$.

Thus, the fv-set

$$S = (3 \ 3 \ 2 \ 1 \ 1 \ 0), \text{ where}$$

$$f(S,4) = 0$$

$$f(S,3) = 4$$

$$f(S,2) = 4$$

$$f(S,1) = 5$$

is recorded as the APL vector 5 4 4 0.

Two fv-sets can be joined to produce a third in one APL statement, based on the component-by-component addition of vectors. The comparison theorem is most easily applied in this form, as well.

(2) The collection of fv-sets associated with a node.

Each node is associated with 3 shape-sets, one each for the codeable shapes: \square , r , c . Matrix $\text{SHAPESET}[I;j]$ holds the "tag" of node I 's j th shape-set. Each shape-set only holds one fv-set--either the single 0-set fv-set, for shapes r and c , or the fv-set whose single member is the "value" of the node I , $n(I)$, for shape \square . These are the only distinct fv-sets which need representation at each node. \square 's fv-set may be selected as part of shape-set r or c , and treated as the only 1-set member. After all fv-sets of a node are generated and compared, the surviving fv-sets are placed in table $\text{FVSET}[I;]$. $\text{SHAPESET}[I;J]$ holds the index K in FVSET of the single fv-set which is represented in shape-set J of node I . $\text{FVSET}[K;]$ holds the 4-element vector representing that fv-set.

(3) The parse-tree.

A parse-tree structure is represented internally by "downward" pointing links, as well as by node order. Node I of the parse-tree is associated with a vector, $SONS[I;J]$, giving the node number of node I 's J th son. $OPR[I]$ gives the code for node I 's operator. Nodes are numbered such that each entry of $SONS[I,J] < I$. As a result, we can visit nodes of the parse-tree in increasing order of node number with assurance that each node's descendants have been visited before that node.

(4) Tags.

Each fv-set, when stored in $FVSET[I;]$, is associated with tags, giving the $FVSET$ indices of the fv-sets from which fv-set I was created. In addition, other information about the fv-set is stored in the same array.

F. Global Tables

Node N of the expression's parse-tree is associated with:

$SON[N;I]$ = the number of the I th son of node N

$OPR[N]$ = the "operator" of node N

$SHAPESET[N;S]$ = the $FVSET$ index of the single surviving member of the shape-set S of node N . This fv-set is a member of 0-set of the shape-set if S is 'r' or 'c'. When S is 'r' or 'c', 1-set is given by $SHAPESET[N;OMEGA]$.

Fv-set I is associated with:

$TAG[I;IALG]$ = the number of the KEPT which generated fv-set I , that is, the KEPT forming the base of the alg-tree whose fringe-set I represents.

$TAG[I;INODE]$ = the node of one of whose shape-sets I is a member.

$TAG[I;IVAL]$ = the value of fv-set I , i.e., $\max(FVSET[I;J])$

$TAG[I; ISAM]$ = Used during the output-phase to indicate which node of I's fringe-set the result-set of the alg-tree can agree with.

$MATR[I]$ = During the output-phase, holds the intermediate array number of the array which is assigned to hold the result-set of the alg-tree's algorithm.

$TAG[I; ISON+K]$ = the FVSET index of the Kth "son" (generating fv-set) of fv-set I.

$FVSET[I; V]$ = holds fv-set I's $f(I, V)$ value.

The Ith KEPT read in by KEPTS is associated with:

$KEPT[I; 1]$ = the number of the node in the KEPT-forest representing I's root.

$KEPT[I; 2]$ = the root-shape of KEPT I.

$KEPT[I; 3]$ = the number of leaves of KEPT I.

The "forest" of KEPT's stores all nodes of all KEPT's. Each node is assigned a number distinct from the numbers assigned any other KEPT's nodes by "relocating" the numbers assigned nodes on input. A given node, I, of this forest is associated with the following information.

$SONE[I; K]$ = the forest node number of the Kth son of node I.

If I is a leaf, $SONS[I; K] = 0$.

$OPRE[I]$ = the operator of node I, where I is a leaf. If I is a leaf of some KEPT, $OPRE[I]$ gives I's leaf-shape.

F. Description of program operation**TREE**

This "main routine", keyboard activated, accepts an expression's parse-tree, using INTREE. It initializes and structures the arrays needed, and calls ASSIGN to initiate phase 1. On return, it prints the number of intermediate matrices ASSIGN finds to be needed, and calls ALGOR to collect and print the alg-tree assigned.

```

      VTREE[0]V
    V TREE
[1]  INTREE 0
[2]  NODES←pF
[3]  MXSH←3
[4]  LEAFSET← 1 1 1
[5]  MXFVS←MXSH×NODES
[6]  MXVL←4
[7]  IALG←1
[8]  INODE←2
[9]  IVAL←3
[10] ISAM←4
[11] ISON←4
[12] MXTG←ISON+MXLV
[13] SHAPESET←(NODES,MXSH)ρ0
[14] FVSET←(MXFVS,MXVL)ρ0
[15] TAG←(MXFVS,MXTG)ρ0
[16] FVSL←1
[17] XPSS← 1 1 2 1 3 1
[18] SON←S
[19] OPR←O
[20] OMEGA←1
[21] ASSIGN
[22] MATR←MXFVSρ0
[23] FVR←SHAPESET[NODES;1]
[24] VALUE←TAG[FVR;IVAL]
[25] AVAIL←VALUEρ0
[26] T←('ASSIGNED ';VALUE;' MATRICES:')
[27] ALGOR FVR
    V

```

EEPTS

This "main routine", keyboard activated, reads as many EEPT's as the user cares to supply. All are recorded in the same tables, SONE, to hold the "sons" tree representation, and OPRE, to hold the trees' operators. Each EEPT occupies a different (contiguous) set of indices in these tables, with the relocation argument of INTREE used to adjust the values stored into a true list structure. A vector EEPT[I;] holds 3 items of information about EEPT I: (1) the index in SONE and OPRE of its root; (2) its root-shape; (3) the number of leaves it has.

```

      VEEPTS[[]]V
    V EEPTS;Q
[1]  FEPT←10
[2]  MXLV←0
[3]  SONE← 0 0 ρ0
[4]  OPRE←10
[5]  FEP2:INTREEρOPRE
[6]  →FEP1×11≥ρ,0
[7]  FEP3:OPRE←((ρOPRE)+ρ0)ρ(OPRE,0)
[8]  SONE←(((ρSONE)[1]+(ρS)[1]),(ρS)[2])ρ((,SONE),(,S))
[9]  LVS←+/^/[2]S=0
[10] MXLV←MXLV LVS
[11] FEPT←FEPT,(ρOPRE),F[ρF],LVS
[12] →FEP2
[13] FEP1:FEPT←(((ρFEPT)+3),3)ρFEPT
    V

```

ASSIGN

ASSIGN implements the leaves-in-algorithm, as described. ASSIGN visits each node of the parse-tree, T, in order of their node-numbers, and tries to match each KEPT with that node. When a matching KEPT is located, NEWFVS is used to update B, BT, NV, and NT, the "surviving" best fv-sets. After all KEPT's have been tried, the surviving fv-sets are copied into FVSET and TAG. SHAPESET gives their indices, or is 0 for empty shape-sets.

```

      VASSIGN[[]]V
    V ASSIGN;N;F;TST;ISET
[1]  N←1
[2]  FVSL←1
[3]  ASG4:→ASG1×1^/SON[N;]=0
[4]  F←1
[5]  NT←0
[6]  BT←(MXTG,MXSH)ρ0
[7]  B←(MXVL,MXSH)ρ0
[8]  NV←1/10
[9]  ASG2:F NEWFVS N MATCH EEPT[F;1]
[10] F←F+1
[11] →ASG2×1F≤(ρEEPT)[1]
[12] BT[(NV≤[1]B)/1MXSH]←0
[13] B[1]←FVST NV
[14] BT[1ρNT;1]←NT
[15] TST←BT[ISON+1;]≠0
[16] BT[INODE;]←N
[17] ISET←TST/1MXSH
[18] FVSET[FVSL+1ρISET;]← 2 1 QB[;ISET]
[19] TAG[FVSL+1ρISET;]← 2 1 QBT[;ISET]
[20] SHAPESET[N;]←TST\FVSL+1ρISFT
[21] FVSL←FVSL+ρISET
[22] →ASG3
[23] ASG1:FVSL←FVSL+1
[24] TAG[FVSL;IMODF]←N
[25] SHAPESET[N;]←LEAFSET×FVSL
[26] ASG3:N←N+1
[27] →ASG4×1N≤NODES

```

∇

E NEWFVS X

One of the central phase-one routines, NEWFVS generates all fv-sets which match EEPT E at some node N of the expression parse-tree, T. X describes each node of T matching a leaf of E. NEWFVS generates combinations of fv-sets which are members of the proper shape-set of the matching nodes by using the base-2 representation of a number it increments from 0 to 2^pX . When a position of this vector is 0, it selects the single 0-set member of the shape-set. When 1, it selects the 1-set member. A translation vector, XPSS, translates 1-set requests into Ω -shape-set requests, since 1-sets are represented only implicitly. Various tests exclude incorrect combinations, such as 1-set of Ω -shape-set, or requests for an empty shape-set. Each generated combination is JOINed, and tested against the previously-surviving best fv-sets of the shape-set whose name is root-shape(E). Tags, including values of the fv-sets, and "ISAM" are computed here. ISAM designates one node (by fv-set number) whose access-shape agrees with root-shape(E), and which can consequently be assigned a 2-array which is also assigned to hold the result of the algorithm rooted at this node. The principle outputs of NEWFVS are:

$B[;R]$ gives the fv-set surviving in shape-set R, $R \neq \Omega$

$BT[;R]$ gives the tags of $B[;R]$

NV is a scalar, holding $N(S^*)$ of the fv-set S^* of smallest value in shape-sets of this node.

NT holds a copy of the tags of that fv-set whose value appears in NV.

```

VNEWFVS[[]]V
V E NEWFVS X;M;I;FVS
[1] M←(ρX)[1]
[2] →0×1M=0
[3] MD←2×MωM
[4] ML←x/MD
[5] I←0
[6] NF1:MC←MD÷I
[7] →NF2×1∨/(X[;1]=OMEGA)∧MC=1
[8] FVS← 1 1 SHAPESSET[X[;2];XPSST[MC←-1+2×X[;1]]]
[9] FVS←FVS×1-2×(FVS≤0)∧X[;1]=OMEGA
[10] →NF2×1∨/FVS≤0
[11] C←JOINS FVS
[12] SM←(X[;1]=OMEGA)∧MC=1
[13] SMI←SM11
[14] R←EEPT[E;2]
[15] G←(∨/SM)∧R=OMEGA
[16] SMS←0
[17] →NF6×1G=0
[18] SMS←FVS[SMI]
[19] SMT←TAG[SMS;ISAN]
[20] →NF6×1SMT=0
[21] SMS←SMT
[22] NF6:CV←[ /C
[23] →NF3×1(C[1]=CV)∨G=1
[24] →NF5×1CV≥NV
[25] B[;R]←C
[26] BT[1M+ISON;R]←E, 0 0 ,SMS,FVS
[27] CV←CV+1
[28] NF3:→NF5×1CV>NV
[29] NV←CV
[30] NT←(-E),0,NV,SMS,FVS
[31] NF5:I←I
[32] NF2:I←I+1
[33] →NF1×1I<ML

```

V

N MATCH E

Here N is a node number in the expression parse-tree, and E gives the root of an EEPT. The value of MATCH is an APL matrix, Z. Z[I;] describes the node matching E's Ith leaf. Z[I;1] gives the leaf-shape; Z[I;2] the expression parse-tree node's number. The recursion is performed by MATCH1. MATCH restructures MATCH1's result, which is an APL vector, into the more convenient form of an APL matrix.

```

      ∇ MATCH[[]]∇
      ∇ Z←N MATCH E;W
      [1] W←N MATCH1 E
      [2] Z←(((ρW)+2),2)ρW
      [3] →0×1∧Z[;2]≠0
      [4] Z←(0 2)ρ0
      ∇

```

N MATCH1 E

Recursively matches node N of the parse-tree with node E of an EEPT. Its value describes the list of nodes of the parse-tree which match leaves of the sub-EEPT rooted at E.

```

      ∇ MATCH1[[]]∇
      ∇ Z←N MATCH1 E;I;J
      [1] Z←10
      [2] →0×1E=0
      [3] →MTC1×1N>0
      [4] MTC3:Z← 0 0
      [5] →0
      [6] MTC1:Z←OPRF[F],N
      [7] →0×1∧/SONE[E;]=0
      [8] →MTC3×1OPRF[F]≠OPR[N]
      [9] J←(ρSON)[2]
      [10] I←1
      [11] Z←10
      [12] MTC2:Z←Z,SON[N;I]MATCH1 SONE[E;I]
      [13] I←J+1
      [14] →MTC2×1I≤J
      ∇

```

A JOIN B

A and B are fv-sets represented by APL vectors. The value of 'A JOIN B' is the APL vector representation of fv-set C, where $C = A \cup B$.

```

      ∇ JOIN[ ] ∇
      ∇ C ← A JOIN B
[1]  C ← A + B - (0 ≠ A × B) × (1 ρ B) - 1
      ∇

```

JOINS X

X is a vector of fv-set indices. The value of JOINS X is JOIN/FVSET[X;], if this could be written in APL, i.e., an fv-set vector representing $\bigcup_I \text{FVSET}[X[I];]$.

```

      ∇ JOINS[ ] ∇
      ∇ V ← JOINS X;I
[1]  I ← 1
[2]  V ← 0
[3]  JNS1: V ← V JOIN FVSET[X[I];]
[4]  I ← I + 1
[5]  ~JNS1 × 1 I ≤ ρ X
      ∇

```

FVST X

X is a scalar. FVST X has as value an APL-vector representation of the single-integer fv-set (X).

```

      ∇ FVST[ ] ∇
      ∇ Z ← FVST X
[1]  Z ← (1 X), (MXVL - X) ω 0
      ∇

```

Output phase.

Once each node has been visited, and its shape-sets and their member fv-sets have been computed, the list of alg-trees represented must be produced. In the course of visiting each node in the expression's parse-tree, each node has been "evaluated". Furthermore, "tags", tracing the ancestry of each fv-set have been recorded. These tags represent alg-trees in a true "sons" tree representation. The output-phase proceeds from the root-node of the parse-tree to the leaves, collecting each alg-tree, ordering its fringe-set, and collecting the alg-trees rooted at each of the fringe-set nodes. Recursion reverses the printing order so that alg-trees rooted at fringe-set members of alg-tree A print before A. Intermediate 2-arrays are assigned "linearly". Each 2-array is given an "available" indicator. As each alg-tree is printed, an available 2-array's assigned for its result (root-node), and the 2-arrays assigned its fringe-set nodes are made available.

Output routines:

ALGOR X

X gives the FVSET index of an fv-set which is the "root" of an alg-tree. ALGOR collects the fringe-set fv-sets, S[I], using COLLECT, orders them by fv-set value, using ORDER, then calls itself on each of the S[J[I]] to compute and print the alg-trees which are rooted at each input to X's alg-trees. It then prints X's alg-tree.

```

      VALGOR[[]]▽
    ▽ ALGOR X;F;G;I
[1]  I←TAG[X;IALG]
[2]  →ALG3×1I<0
[3]  ERROR IN TAG
[4]  ALG3:TAG[X;IALG]←-I
[5]  F←COLLECT X
[6]  G←ORDER F
[7]  I←1
[8]  →ALG2
[9]  ALG1:ALGOR F[G[I]]
[10] I←I+1
[11] ALG2:→ALG1×1I≤pG
[12] PRT F
[13] TAG[X;IALG]←0
    ▽

```

COLLECT X

The value of COLLECT X is a vector, listing all fv-sets in the connection-set and fringe-set of the alg-tree rooted at the fv-set whose index is X. An fv-set in the fringe-set is identifiable because each such fv-set Y is the "value" of some node, and is marked during phase 1 by recording a negative TAG[Y;IALG] entry for it. Also, only these fv-sets have non-zero TAG[Y;IVAL] entries.

```

      VCOLLECT[[]]V
    V Z←COLLECT X;A;I
  [1] Z←,X
  [2] A←TAG[X;IALG]
  [3] +0×1A≤0
  [4] I←EEPT[A;3]
  [5] COL1:Z←Z,COLLECT TAG[X;ISON+I]
  [6] I←I-1
  [7] +COL1×1I>0
    V

```

ORDER X

Has as value a vector, Y, which lists certain indices in vector X. Y satisfies: TAG[X[Y[I]];IVAL] ≥ TAG[X[Y[I+1]];IVAL]

and TAG[X[Y[I]];IVAL] > 0

Thus, according to our rule for ordering the computation of a fringe-set, fv-sets X should be computed in the order given by Y. Note that only fringe-set members of X are listed in Y. Fv-sets X[I] which are in the connection set of an alg-tree, or are associated with leaves of the parse-tree, have zero TAG[X[I];IVAL] entries, and hence are not listed in Y.

```

      VORDER[[]]V
    V C←ORDER Y;T;V
  [1] C←10
  [2] V←TAG[Y;IVAL]
  [3] V[1]←0
  [4] +ORD2
  [5] ORD1:T←V[1]/V
  [6] V[T]←0
  [7] C←C,T
  [8] ORD2:→ORD1×1V/V>0
    V

```

PRT X

PRT prints the alg-tree whose root fv-set, connection fv-sets and fringe fv-sets are listed, represented by their FVSET indices, in vector X. PRT also assigns intermediate 2-array numbers to the root fv-set's node, and frees the 2-arrays assigned the fringe fv-sets, using NEWM and FREEM.

```

      VPR[T[[]]7
    V PRT X;Y
  [1]  [+('NODES: ';TAG[X;INODE])
  [2]  [+('EEPTS: ';TAG[X;IALG])
  [3]  Y←X[1]
  [4]  SAM←TAG[Y;JSAM]
  [5]  →PR1×1SAM=0
  [6]  MATR[Y]←MATR[SAM]
  [7]  →PR2
  [8]  PR1:MATR[Y]←NEWM
  [9]  PR2:FREEM MATR[X]
  [10]  [+('TEMPS: ';-MATR[X])
  [11]  [+10
      V

```

NEWM

NEWM's value is the index of the first "available" intermediate 2-array. NEWM also sets that 2-array unavailable.

```

      VNEWM[T[[]]7
    V Z←NEWM
  [1]  Z←AVAIL10
  [2]  AVAIL[Z]←1
  [3]  Z←-Z
      V

```

FREEM X

Frees (makes available) intermediate matrix X.

```

      VFREEM[[]]V
    V FREEM X;I;T
[1]  I←1
[2]  →FR2
[3]  FR1:T←-X[I]
[4]  →FR2×1T≤0
[5]  AVAIL[T]←
[6]  FR2:I←I+1
[7]  →FR1×1I≤pX
    V

```

Input subroutine**INTREE T**

INTREE accepts a parse-tree or KEPT from the keyboard. Its output is the "FATHERS" vector (in F), the "OPERATORS" vector (in O), and a generated matrix of sons (in S), giving the "reverse" links of the FATHER vector. Its single argument, T, is used to "relocate" the list structure produced in S and F, so that the value actually stored in F satisfies $F[I] = T+J$, where node J is node I's father, according to the input structure.

```

      VINTREE[[]]V
    V INTREE T;I;J;M
[1]  [←'FATHERS'
[2]  F←[
[3]  O←F
[4]  →O×11≥p,F
[5]  MXSNS+2
[6]  S+((pF),MXSNS)p0
[7]  O←(pF)p1
[8]  I←1
[9]  INT2:J←F[I]
[10] S[J;O[J]]←I+T
[11] O[J]←O[J]+1
[12] F[I]←J+T
[13] I←I+1
[14] →INT2×1I<pF
[15] [←'OPERATORS'
[16] O←[
[17] J←O=F
    V

```


BIBLIOGRAPHY

1. Conway, R. W., and H. L. Morgan, "Tele-CUPL: A Telephone Time Sharing System," *Comm. ACM* 10, 9 (September 1967), 538-542.
2. Falkoff, A. D. and K. E. Iverson, "The APL/360 Terminal System," Research Report RC-1922, IBM Watson Research Center, Yorktown Heights, N.Y. (October 1967).
3. Floyd, R. W., "Assigning Meanings to Programs," *Proc. of Symposium in Applied Mathematics, Mathematical Aspects of Computer Science*, Volume 19, American Mathematical Society, 1967.
4. Galler, B. A., and A. J. Perlis, "Compiling Matrix Operations," *Comm. ACM* 5, 12 (December 1962), 590-594.
5. Nakata, I., "A Note on Compiling Algorithms for Arithmetic Expressions," *Comm. ACM* 10, 8 (August 1967), 492-494.
6. Naur, P. (editor), et. al., "Revised Report on the Algorithmic Language ALGOL 60," *Comm. ACM* 6, 1-17 (1963).
7. Reinwald, L. T., and R. M. Soland, "Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time," *J. ACM* 13 (1966), 339-358.
8. _____, and _____, "Conversion of Limited-Entry Decision Tables to Optimal Computer Programs II: Minimum Storage Requirement," *J. ACM* 14, 4 (October 1967), 742-757.
9. Simon, H. A., "Experiments with a Heuristic Compiler," RAND Corp. P-2349, June 30, 1961.
10. Walker, R. J., "An Instruction Manual for CUPL, The Cornell University Programming Language," Cornell University, December, 1966.
11. Winograd, S., "On the Time Required to Perform Addition," *J. ACM* 12, 2 (April 1965), 277-285.
12. _____, "On the Time Required to Perform Multiplication," *J. ACM* 14, 4 (October 1967), 793-802.
13. _____, "On the Number of Multiplications Required to Compute Certain Functions," *Proc. Nat. Acad. Sciences*, 58, 5 (November 1967) 1840-1842.
14. _____, "A New Algorithm for Inner Product," IBM Watson Research Center, Yorktown Heights, New York, research report RC-1943, November 21, 1967.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP	
3. REPORT TITLE SOME TECHNIQUES FOR ALGORITHM OPTIMIZATION WITH APPLICATION TO MATRIX ARITHMETIC EXPRESSIONS			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Scientific Interim			
5. AUTHOR(S) (First name, middle initial, last name) Robert A. Wagner			
6. REPORT DATE 27 June 1963		7a. TOTAL NO. OF PAGES 159	7b. NO. OF REFS 14
8a. CONTRACT OR GRANT NO. FH4620-67-C-0053		8b. ORIGINATOR'S REPORT NUMBER(S)	
9. PROJECT NO. 9718			
10. OTHER REPORT NO(S) (Any other numbers that may be assigned) 614450 IF 691304		AFOSR 68-2641	
11. DISTRIBUTION STATEMENT 1. This document has been approved for public release and sale; its distribution is unlimited			
12. SUPPLEMENTARY NOTES TECH, OTHER		13. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Research (SRMA) 1400 Wilson Boulevard, Arlington, Virginia, 22202	
14. ABSTRACT Algorithm optimization can be accomplished by an exhaustive search over alternative algorithms for performing some programming task. The resulting algorithms are optimum only with respect to a program technology--the particular set of alternatives investigated. Thus, larger program technologies can be expected to yield better algorithms. This thesis contributes to the production of optimum algorithms in two ways. First, a technique ("loop-fusion") was developed for producing new algorithms equivalent to old algorithms, and thus expanding program technologies. Second, a technique ("comparison") is described which reduces the effort required by certain exhaustive searches over "well-structured" search spaces. These techniques are applied to the production of algorithms for evaluating matrix arithmetic expressions (MAE). (The operators, + and *, in such arithmetic expressions are interpreted as matrix addition and multiplication, respectively.) A method is described for producing, for any MAE, an algorithm for its evaluation which requires fewest arrays for holding N by N matrices, while not requiring more execution time than the "standard" MAE evaluation algorithm. Although the algorithm-production method used is basically an exhaustive-search over a large space of program alternatives for each subexpression of the given MAE, the effort this method requires grows only linearly with the number of operators in the given expression.			

DD FORM 1473
1 NOV 62

Security Classification

Security Classification

14.	KEY WORDS	LINK A		LINK B		LINK C	
		ROLE	WT	ROLE	WT	ROLE	WT

Security Classification